Lattice

Debugger

Debugger

Lattice® CodePRobe

Version 1.0

A Source-Level Debugger for the Lattice C Programming Environment

Lattice, Incorporated
2500 S. Highland Avenue
Lombard, IL 60148
USA

Subsidiary of SAS Institute Inc.

Lattice® CodePRobe User's Manual

This document was produced using HighStyle™, the Lattice Document Composition System.

SAS/C® Debugger

Version 1.0

A Source-Level Debugger for the SAS/C® Programming Environment

SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513-2414
USA

SAS/C® Debugger User's Manual

This document was produced using ***HighStyle***®, the Lattice Document Composition System.

Table of Contents

APPENDICES

Section 1

Presenting CodePRobe

CodePRobe is a source-level debugger that allows you to examine the behavior of a program written in the C language using the syntax and semantics of C. Any variable can be accessed by name as defined within a C module. Data are always displayed in a form consistent with its type: characters are displayed as ASCII text if possible, while floating point values and integers are always displayed as declared. You can reference all complex data types supported by the C language including structures, unions, arrays, members of structures and unions, array elements, bit fields, enumerators, and pointers.

CodePRobe allows you to run a program in a controlled environment where execution can be stopped at any point. As a program is executed under debugger control, the associated source is always displayed in a window. By a simple double click of a mouse button, breakpoints can be set in terms of source file line numbers. More complex conditions for breakpoints are also supported. A powerful "watch break" facility allows you to have the program break execution whenever a variable changes value.

Unlike "symbolic" debuggers, **CodePRobe** can completely isolate the programmer from machine language and assembly-level debugging if desired. However, for those who do wish to interface both C and assembly modules, **CodePRobe** has many features for assembler language support. Registers

can be displayed and set, and instructions can be disassembled and breakpointed in the source window.

This chapter briefly explains what **CodePRobe** does, how it works, and what's involved in debugging a program. It presents the organization of the manual, describing sections which compose the manual. It also discusses notational conventions associated with words and syntax used throughout the manual. Throughout this manual, we use the terms **CodePRobe** and **CPR** interchangeably to refer to this debugger program.

1.1 Overview of the Program

CodePRobe implements a number of commands and features specifically designed to ease the debugging of C programs. In addition to a command-line interface, **CPR** provides you with a sophisticated, screen-oriented interface where you may take advantage of multiple windows and pull-down menus. Plenty of shortcuts are built into the program as well, enabling you to issue commands with a couple keystrokes or mouse clicks.

As a source-level debugger, **CodePRobe** allows you to:

- single step through programs
- set breakpoints on C source lines
- examine variables and code at the C language level
- display all C data types–structures, unions, arrays, enumerators, and bit fields–in their C formats
- display hex dumps of memory
- continuously watch any C variable, array elements, and structure members
- assign a value to a C variable
- fill an area of memory with values according to the type of objects occupying that memory
- copy one C array or structure to another

- copy a string from one place to another
- view and manipulate register data
- access all storage classes of data, including automatic, static, and register as well as external.

CodePRobe allows you to debug programs that incorporate the Amiga's multi-tasking features. Programs may call the EXEC library function *AddTask* while under debugger control. Breakpoints may be set and trapped for any task generated under debugger control. The user can choose to display the state of any task in a multi-tasking application, including the stack and registers. Tasks can be selectively started and stopped. Additionally, tasks that are executed independent of the debugger can be intercepted and attached to the debugger. This can be useful if a process enters an infinite loop or waits indefinitely on a message port.

1.1.1 How CodePRobe Works

A program can be run under debugger control by typing "cpr" followed by the command line you'd normally use for the program. To display the C source symbols and attributes of your program, however, you must take certain steps when you compile and link the program. **CodePRobe** requires that the compiler generate additional debugging information in the final executable module.

The Lattice C Compiler supports several options pertaining to the amount of debugging information that is passed on to the linker. The Lattice linker, **blink**, also requires debugging options in order to (1) pass on information about libraries that have been linked into the executable module, and (2) allow a user to strip out debug information.

1.2 Lattice AmigaDOS 68000 Macro Assembler

By using the command "set source asm", you can direct **CPR** to disassemble code in the source window. **CPR** can single-step (trace) and step over (proceed) assembler instructions as well as C source. Also, absolute addresses

and registers can be referenced when displaying data as well as symbolic addresses. The Register Window is ideally suited for use with assembler.

1.2.1 How CodePRobe is Controlled

CPR is a full-screen debugger. A user can control **CPR** by a number of means. Among **CodePRobe**'s controls are pull-down menus, Intuition-based windows, and mouse support–features similar to other popular Workbench applications. **CPR** also has powerful, highly specific commands which can be entered by means of the Dialog Window. In addition, **CPR** offers a number of keyboard shortcuts, special functions, and menu accelerators to maximize user efficiency.

1.2.2 Debugging a Program

Debugging a program using **CPR** involves three basic steps:

1. Compile with debug information:
   a. select compiler options (the -d options of the LC command)
   b. select linker options (use the ADDSYM option to get all external symbols)
2. Invoke debugger
3. Execute program under debugger control

Of course, there are other steps involved in setting up a debugging environment. Section 3 goes into greater detail on this topic.

1.3 What's in the Manual

In little time, you will become comfortable with **CPR**'s use. This manual complements the ease of the program by providing step-by-step examples and sample screen illustrations, along with detailed instruction about all aspects of the program.

This manual is primarily oriented toward programmers already familiar with

the C language. Discussion of the assembly-level tools, such as the Register Window, assumes familiarity with the 68000 processor.

The manual is divided into 10 sections and an appendix as follows:

- Section 1 introduces the program, describes the organization of the manual, and sets forth notational conventions.
- Section 2 reviews Workbench/Intuition concepts, examines the layout of a sample screen, and walks through a sample debugging session.
- Section 3 tells you how to prepare your program for **CPR**, how to determine source file location, and what command-line options are available, how to use the following commands: **quit**, **set**, and **show**.
- Section 4 describes how to control **CPR** through pull-down menus, special function keys, accelerator keys, and dialog commands.
- Section 5 details five types of commands associated with examining data: **display**, the **dump** commands, **register**, **register flag dump and set**, and **whatis**.
- Section 6 introduces four commands used to modify code or data: **enter**, **fill**, **memcpy**, and **strcpy**.
- Section 7 describes how you can control program execution with the commands **go**, **proceed**, **trace**, **return**, **restart**, and the **break** commands, and introduces the concepts of single stepping and breakpoints.
- Section 8 presents the commands used to control watches and watch breaks, giving numerous examples of how these commands are executed and interpreted.
- Section 9 describes specialized commands associated with the program, including **list**, **unassemble**, **execute**, and **hunks**.
- Section 10 discusses multi-tasking by introducing the following commands: **tasks**, **set task**, **deactivate**, **activate**, **detach**, **catch**, and **symload**.

There is additional information in the **READ.ME** file on the first disk of your compiler package. Always check the **READ.ME** file provided on this disk for information on bug fixes, problems, and errata. You can access this file through the **READ.ME** icon or you can use the AmigaDos type command to

obtain a listing of this file. Another alternative is to use an editor such as the Lattice Screen Editor to view the file.

If you need additional help, we also offer the Lattice Bulletin Board Service (LBBS), a multi-user bulletin board system for all owners of Lattice products. LBBS may be reached via modem by dialing (312) 916-1200, setting your communication parameters to 300-2400 baud, 8 data bits, 1 stop bit, and no parity.

1.4 Matters of Style

To make this manual more useful for you, we've incorporated a number of its special features including distinctive fonts, special icons, and complex tables. In order to make it easier for you to follow the descriptions of **CodePRobe** commands, modes, windows, and menus, a number of notational conventions are used in this manual.

bold — **CPR** commands are referred to in boldface, as are command line options to the compiler, linker, and debugger.

italics — File names and program variables (including function names) appear in italics.

parentheses ( ) — In sample syntax, parentheses in **bold print** indicate that these symbols should be typed in literally as shown. This will also be noted in the text.

square brackets [ ] — The usual convention of denoting arguments, selections, parameters, and portions of command names as optional is to enclose them in square brackets: [ ].

vertical bar | — Alternatives are indicated by means of the vertical bar: |. For purposes of clarity and grouping, such alternatives may be enclosed within parentheses, but these should not be typed in explicitly. Thus, the line:

```
command (a1 | a2 | a3) [(a5 | a6)]
```

describes a command having one required argument,

which may be either a1, a2, or a3 and one optional argument, which may be either a5 or a6. More detailed examples of such conventions are given in Section 4.

`font change` A monospaced Courier font is used for examples of source code and command-line input.

Section 2

Looking at CodePRobe Windows

From the start, Amiga users should feel comfortable with **CodePRobe** because it has many of the same features associated with other Amiga applications. The program uses overlapping windows and pull-down menus to create a powerful and easy-to-use debugging environment.

Before we describe the specific windows and features that make **CPR** an exceptionally useful debugger, however, we will review some of the basics of the Amiga work environment. If you are already familiar with Workbench/Intuition, you may wish to skip over this part.

2.1 Workbench/Intuition Windows and Screens

The advantage of the Amiga's Workbench is that users can depend on the same "look and feel" when they move from application to application. Mouse buttons also perform in a similar manner. Amiga users will feel immediately comfortable with **CodePRobe** because it operates identically to other Workbench applications. Figure 2.1 shows how **CPR** looks upon starting the program.

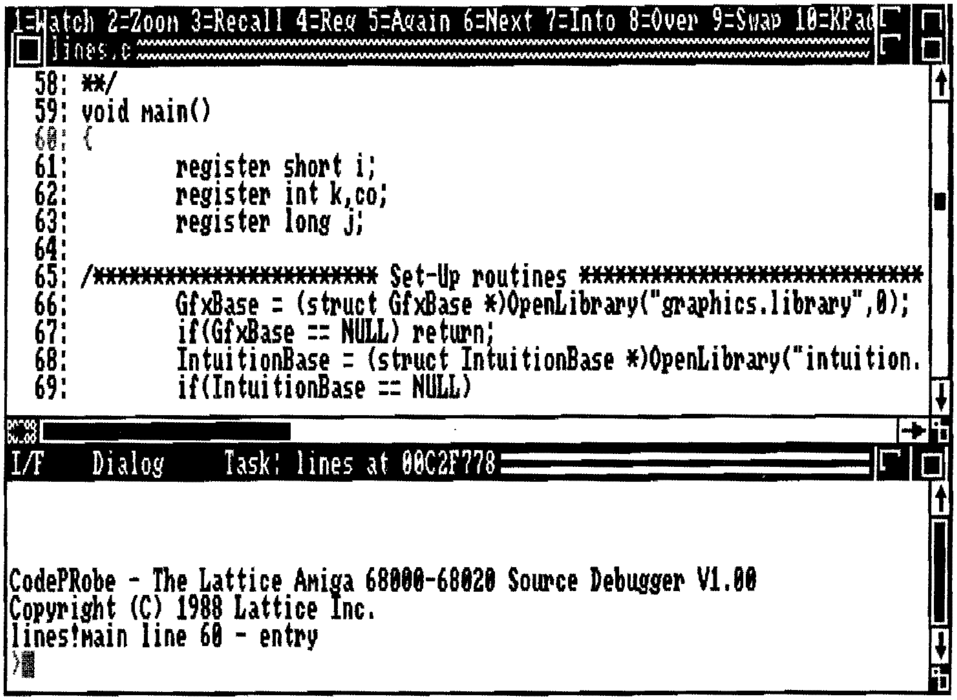

Figure 2.1 Initial CodePRobe Screen

2.1.1 The Title Bar

Most Workbench applications have a title bar,; usually a narrow strip at the top of a screen or window that gives the name of the screen or window. **CPR**'s screen title bar gives a brief description of the function keys as a handy reference. In addition, there are unique title bars for each of its four windows.

2.1.2 The Menu Bar

The menu bar is a strip at the top of the screen. As with other Workbench applications, **CPR**'s menu bar appears in place of the screen title bar whenever you push the menu button (i.e., the right button of the mouse).

The menu bar displays the basic menus available to you. The menu bar of **CodePRobe** offers the following menus from which to choose items:

File	Options	Run	Break	Watch

You can activate the list of menu items associated with each menu by moving the mouse pointer to the menu bar (while pressing the menu button), you can activate the list of menu items associated with each menu. By moving the pointer through the menu, you can then select a menu item by highlighting that item and releasing the menu button. Some menu items have sub-items as well. Figure 2.2 shows an example of **CPR**'s pull-down menus.

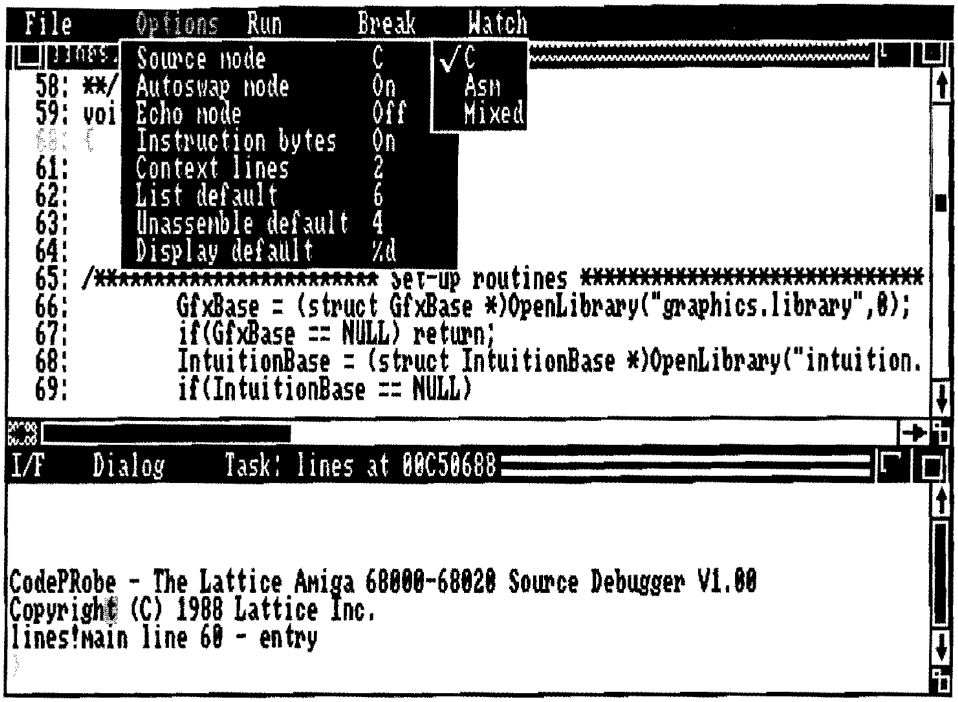

Figure 2.2 Pull-Down Menus

2.1.3 Gadgets

Gadgets are facilities provided within a window, requester, or screen that allow you to change what's being displayed or to communicate with a tool. Types of gadgets include sizing gadgets, front gadgets, back gadgets, and close gadgets. **CPR** makes use of the same familiar gadgets offered in other Workbench applications.

Scroll bars are specialized types of gadgets that allow you to display different parts of a project within a window. Scroll bars may be either vertical or horizontal depending on the design of the window. A scroll bar contains a

scroll box and two scroll arrows, any of which may be used to move a window's contents on the display. The size of the scroll box depends on the proportion of the project which is displayed in the window.

The arrows move up or down through a project *one line at a time*. You can move any distance through a project by sliding the scroll box up or down, or you can click on either side of the scroll box to move through the project *one page at a time*.

2.1.4 Preferences

The Workbench tool Preferences allows you to adjust various settings on your Amiga. For instance, you can use the Preferences tool to customize Workbench colors. Since users can select color settings different from the original Workbench colors, we will avoid referring to any specific colors when describing highlighted text. Instead we will refer to the four Workbench colors (from left to right) as pen 0, pen 1, pen 2, and pen 3.

2.2 Layout of a Sample Screen

CodePRobe provides a sophisticated screen-oriented interface where you may take advantage of multiple windows and pull-down menus. You may use windows to issue **CPR** commands, to receive output from **CPR** commands, or to browse through the source of your program.

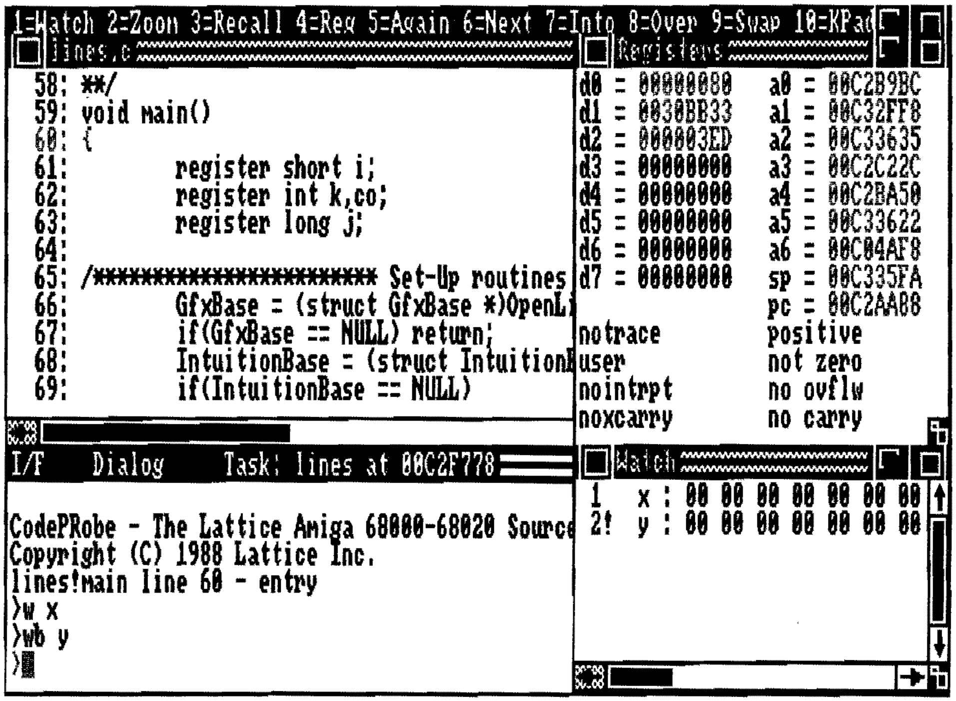

Figure 2.3 CodePRobe Windows

Figure 2.3 is a sample **CodePRobe** screen showing all the major components of a fully outfitted screen including:

- Source Window
- Dialog Window
- Register Window
- Watch Window
- All associated gadgets

When the menu button on the mouse is pushed, the screen title bar and menu bar appear on the screen.

CPR has four basic windows: the Dialog Window, the Source Window, the Register Window, and the Watch Window. One of the four window is called the *active window* because it is the window currently receiving (or waiting for) input. The title bar of the active window is always solid.

The **Dialog Window** allows you to enter commands and review the results of those commands. Many of the same commands available through the pull-down menus can be invoked directly at the command line in the Dialog Window.

The **Source Window** displays the source code for the module currently being executed. As execution proceeds, the Source Window is continually updated to reflect the current position at which the application is running. The Source Window highlights the current position (the line about to be executed) and any breakpoints.

The **Register Window** is an optional window that is most useful to assembly-level programmers. It is also valuable for checking registerized variables pertaining to C source code. Once opened, it displays the current contents of all registers and the current values of the processor flags. Any registers that have been modified since the last breakpoint will be highlighted. The Register Window can be opened by pressing the F4 key.

The **Watch Window** is another optional window. Using the Watch Window, you can display variables of interest. For instance, you can set a watch break such that the application will be stopped as soon as the watch variable changes value. The Watch Window can be opened by pressing the F1 key.

2.3 A Walk Through a Typical Application

The best way to understand how to use **CodePRobe** is to walk through a sample debugging session. In the following example, you will (1) learn to maneuver in **CPR**'s four windows and (2) get a working knowledge of some common commands.

This example will introduce you to a number of commands and **CPR** options. For ease of learning, the discussion is kept simple to avoid excessive detail. Instead, the sections which follow this one will provide all the necessary description to get the most out of **CPR**.

The source code for this sample program can be found on disk 4 of this compiler package in the directory *:examples/debugger.*

Compile and link the program

The sample program, *Nervous Lines*, must be compiled with the **-d3** option (full debugging) of the compiler. This option outputs full debugging information for those symbols and structures referenced by the program. The **-L** option invokes the linker automatically.

```
lc -d3 -La lines
```

When any of the debug options are used, **lc** passes the ADDSYM option to **blink**. The ADDSYM option causes **blink** to emit HUNK_SYMBOL records for all external symbols in the input object files and libraries regardless of whether the input object file was compiled with the **-d** option. If you wish to invoke the linker as a separate step, the following command line should be invoked:

```
blink lib:c.o lines.o to lines lib lib:lc.lib lib:amiga.lib ADDSYM
```

Invoke the debugger

To start the debugger, you type in `cpr` followed by the executable program's filename.

```
cpr lines
```

Note that the Dialog Window and the Source Window open by default. The Dialog Window highlights the current edit line in pen 3. In the same manner, the Source Window highlights the next line of the program's source code to be executed.

Set the mode through a pull-down menu

Display modes can be easily modified by use of a pull-down menu. Hold down the right mouse button and drag the pointer through the menu items under the Options menu. The Source Mode option appears as a menu item

and indicates C as the default mode. When the Source Mode option is highlighted, three sub-items are offered as choices: C, asm, and mixed. Modifying the selected choice alters the appearance of the lines in the Source Window accordingly. For instance, Figure 2.4 shows the results of selecting mixed mode, while Figure 2.5 shows the results of selecting assembly mode.

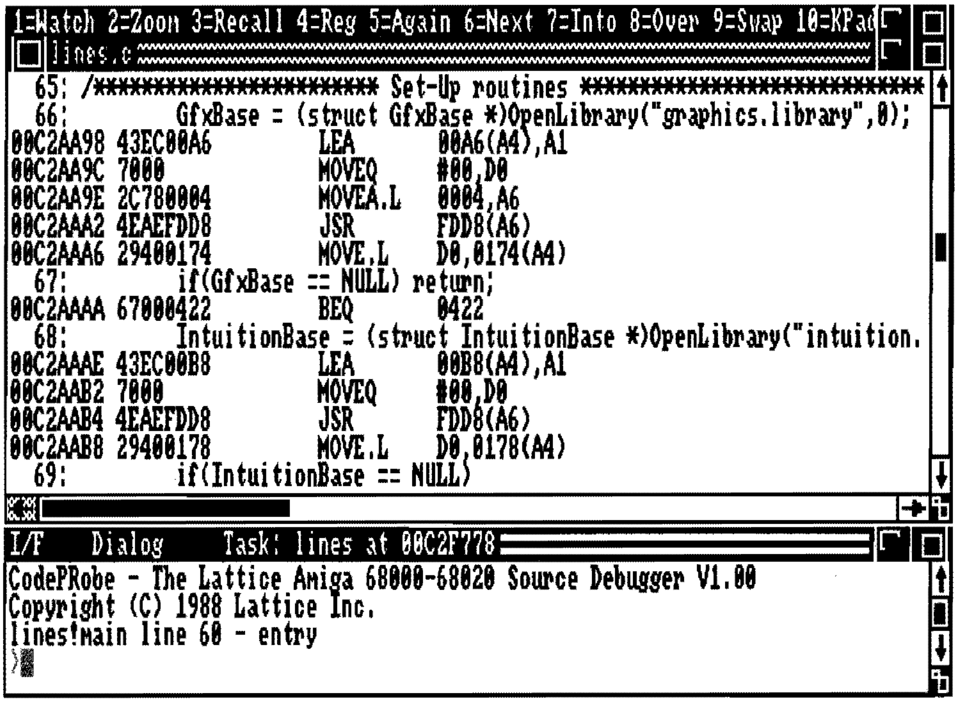

Figure 2.4 Source Window Showing Mixed Mode

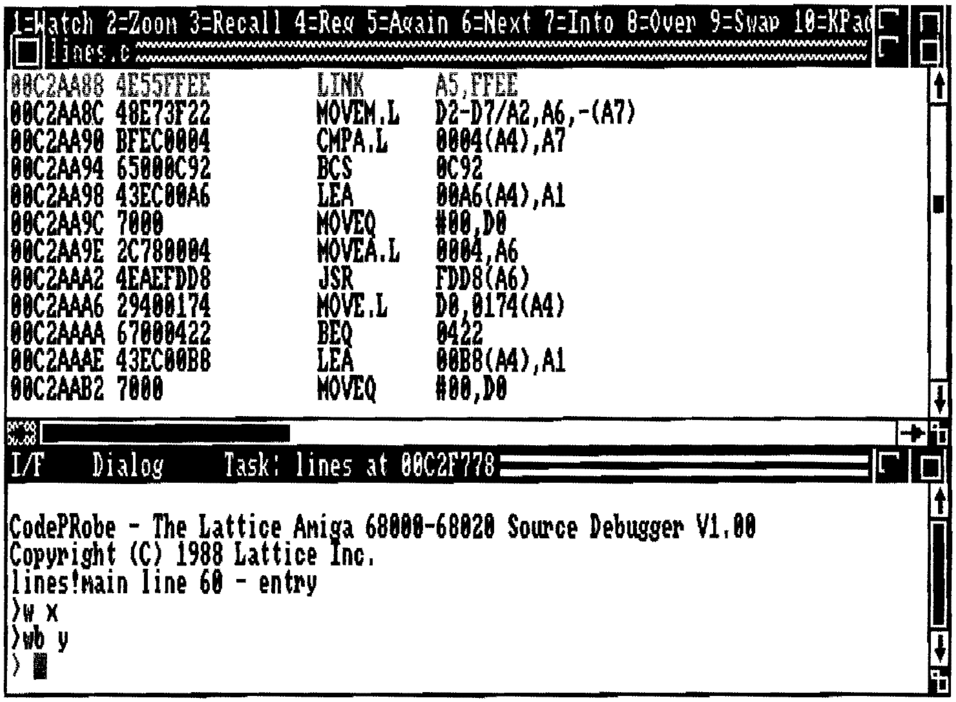

Figure 2.5 Source Window Showing Assembly Mode

Through the Options menu, you can also activate the autoswap mode. When autoswap is on, the screen containing the application's output is pushed to the front each time **CodePRobe** gives control to the application. When a breakpoint is reached, the debugger screen is again pushed to the front.

Step over code

One of most useful features of **CPR** is the ability to step through code. As this is probably the most frequently used command, simply pressing the return key will cause the **proceed** command to be executed. Step over the first few lines of code as follows:

```
lines!main line 60 - entry
>
lines!main line 66
```

```
>
lines!main line 67
>
lines!main line 68
```

Notice the quick flashing due to the display of the application window, (i.e. the initial workbench CLI window) during each step. Select the Options menu and turn off autoswap mode. Now step over several more lines of code.

```
>
lines!main line 74
>
lines!main line 75
>
lines!main line 76
```

You will also notice that **CodePRobe** does not stop at every line. This is due to the fact that code is not generated for all lines. This is generally true for lines that declare variables, such as lines 61-63.

Display a structure or an array

The **display** command allows you to easily display structures and arrays. For instance, to display the structure `nw`, issue the following command in the Dialog Window:

```
>display nw
struct NewWindow {
   LeftEdge = 100
   TopEdge = 100
   Width = 300
   Height = 100
   DetailPen = '\x02'
   BlockPen = '\x01'
   IDCMPFlags = 514
   Flags = 1039
   FirstGadget = 0x00000000
   CheckMark = 0x00000000
   Title = 0x00226028
   Screen = 0x00000000
```

```
BitMap = 0x00000000
MinWidth = 50
MinHeight = 50
MaxWidth = 640
MaxHeight = 400
Type = 1
}
```

Use F2 to zoom the Dialog Window to full size, so that the entire structure can be viewed at once. The value of the character pointer `Title` will probably be different on your screen, but the other values should be the same. When you have finished viewing, press F2 again, to restore the Dialog Window to its original size.

Individual members of the structure can be displayed as well. For example:

```
> d nw.Type
1
```

To display the two-dimensional array `ox`, issue the following command from the Dialog Window:

```
> d ox
{
[0] = {
   [0] = 0
   [1] = 0
   [2] = 0
      .
      .
      .
```

again, press F2 to zoom the Dialog Window to full size. While the values aren't noteworthy, you can get an idea of the format that is displayed. Notice that the entire array does not fit on a full-size window. To see the beginning values, use the mouse and the Dialog Window's vertical scroll gadget to scroll backwards.

Individual members of structures and unions as well as elements of arrays

can also be displayed using **display** (which can be abbreviated as **d**), as in the following examples:

```
> d nw.Type
1
> d x[1]
0
> d w->RPort
0x07F000FC
> d ox[1][5]
0
> d nw.Title
0x00226028
> dz nw.Title
"Nervous Lines"
```

You will probably find different results for the third and fifth examples. Notice that the last command "dz" (display as zero terminated string) indicates that `nw.Title` should be treated as a pointer to a null-terminated string and its contents displayed rather than the pointer itself.

Set a breakpoint

A breakpoint is an address at which program execution stops whenever encountered. To set a breakpoint, move the pointer to a given line of code in the Source Window. Using the left mouse button, double-click on the line. This sets a breakpoint on the line, as indicated by highlighting with pen 2. (If there is no code generated at the line, an error message will be displayed in the Dialog Window.) The breakpoint can easily be removed by double-clicking again on the particular line of code.

Execute up to a breakpoint

With a breakpoint set at line 106, execute the program up to the breakpoint by typing in the **go** command at the command line. The breakpoint line is now highlighted as pen 3 text on pen 2 background. Double-clicking on the line turns off the breakpoint. The line reverts back to pen 3, indicating this is the next line to be executed.

Display the value and type of a variable

CPR will display a variable's value and tell what type it is. Enter the following commands at the Dialog Window:

```
> d i
0
> whatis i
register in d7 (2 bytes)
short
> whatis x
extern at 0022625E (8 bytes with 4-byte elements)
long [2]
> whatis nw
extern at 00226036 (48 bytes)
struct NewWindow
```

Notice that the **whatis** command can provide information as to the location and attributes of any variable. This is particularly useful for keeping track of automatic variables, which may be placed in registers by the compiler, even though they were not explicitly declared as type `register`.

Use the watch and watch break commands

A watch allows you to monitor a variable or area of memory to see when the value of the variable changes or when the area of memory is altered. For instance, you can create a watch for the variable `x` by issuing the following command:

```
> watch x
```

Pop up the Watch Window by pressing the F1 key. Note that the output shown in the Watch Window is a hex dump of the variable `x`:

```
1    x : 00 00 00 00 00 00 00 00
```

The number 1 at the left of the display identifies the watch. It is used by other commands that manipulate the Watch Window.

A watch break is similar to a watch but acts as a breakpoint when the variable or area of memory is changed. You can create a watch break for the variable `y` by issuing the following command:

```
> wb y
```

At this point the watch window should appear as follows:

```
1     x : 00 00 00 00 00 00 00 00
2!    y : 00 00 00 00 00 00 00 00
```

Notice that the watch break is highlighted in pen color 2 and has an exclamation mark following the watch break number. The exclamation mark indicates that it is a watch break and not simply a watch.

Execute until the variable changes value

The Watch Window is updated when control is returned to you by the debugger (such as at a breakpoint) and the new values of the watched areas are displayed. Note that running with watch breaks set can be a slow process because the Amiga's M68000 processor executes the application in trace mode. This means that control returns back to the debugger after every instruction. The debugger must then check the memory for all watch breaks before restarting the application. The watch break feature is a powerful tool, particularly useful for locating obscure bugs that cause memory to be overwritten from unusual places in the code; however, it should not be used indiscriminantly.

To see the effect of the watch break, click in the Dialog Window and issue the **go** command:

```
> go
break (watch #2)
lines!main line 112
```

Notice that the debugger reports the watch break number that caused the breakpoint to occur. Also, notice that the line that modified the variable was actually the previous line. Line 112, is the next line to be executed.

Clear all watches and watch breaks

To clear watches and watch breaks, issue the following Dialog Window command:

```
> wc *
```

The asterisk * is a wild card representing "all watch and watch break identifiers."

Now that all watches and watch breaks have been cleared, issue the execute command **go** to start the program *Nervous Lines*. Notice that no breakpoints can now be reached so the program will continue until completion. Standard input/output is written to the application window in the Workbench screen.

To see the application as it is running, push F9. The demo will run until the close gadget is activated on the *Nervous Lines* application window.

The Dialog Window should now display the following:

```
> go
program completed execution (0)
```

The value in parentheses is the return code from the program.

Quit the debugger

To quit the debugger, issue the command **quit** at the command line, or use the pull-down menu item in the File menu.

2.4 Help

The **help** command can be used to get on-line help regarding other debugger commands and other features of the program. The **help** command can be invoked by typing either of the following:

> **help** *topic_name*
> **?** *topic_name*

where *topic_name* is the item of interest. If *topic_name* is a debugger command, the **help** command provides a brief description, a summary of command syntax, some sample uses, and references to other commands.

For instance, the command `help display` would yield the following information on the **display** command:

```
Display -- Display variables or memory in a C-like format.
d (<variable> | <address>) ['(' <type> | <tag> ')'] [format] [','] ...
  >d i               /* display a variable, array, or structure */
  >d f1\i %04x       /* override the default format             */
  >d mod!f1\i (char) /* override the default type               */
  >d i (struct abc)  /* structure and union types are allowed   */
  >d i (abc)         /* the tag alone may be specified          */
  >d 0xff0 (abc)     /* addresses may be specified              */
  >d i, i (char) %x, cptr[i] (date_time), a[**p->u.b->c]
  >d "element a[", i, "] =", a[i]
     /* multiple variables and string constants are allowed,    */
     /* plus complicated C variables with multiple dereferences */
See also "help dump" and "help dzero" for other ways to display
memory.
```

Note that in the help screen, the sample syntax uses single quotes (') to indicate characters which are entered as part of the command; in this case, open and closed parentheses.

In addition to describing the debugger commands, **help** can provide further information on certain command parameters, editing features, and options. Typing **help** by itself provides a complete list of topic areas.

Section 3

Setting Up the Debugging Environment

To get the most utility from **CodePRobe**, you need to know how to set up the optimal debugging environment for your particular application. This involves choosing the best debugging option to use, determining where to locate source files, and selecting necessary command-line options. This chapter discusses these and other topics pertinent to customizing your debugging environment. In the process, it introduces the following **CPR** commands:

Quit	Exit the debugger program.
Set	Modify current option settings which determine how information is displayed or handled.
Show	Display the current option settings.

In addition to these commands, the section describes the command used to invoke **CodePRobe**: `cpr`. This command is particularly significant when you consider the command-line options associated with its use. Selecting the correct options can well determines whether a debugging session will be a productive and useful one.

3.1 Starting and Stopping CodePRobe

It is a simple matter to invoke the debugger and execute your target program under debugger control. Four commands are used to control this aspect of **CodePRobe**: the command line **cpr**, and the internal commands **quit**, **restart**, and **start**).

When the **cpr** command is invoked, **CPR** makes the following assumptions:

- The program specified on the command line exists in the current directory or in the execution path defined by AmigaDOS.
- The modules making up the program to be debugged have been compiled with one or more of the debugging options discussed earlier and have been linked using the ADDSYM option of the linker.
- Any source files you want to access (e.g., to set breakpoints at source lines) exist in either the current directory, your special source path list, or the directory in which the module was compiled.

The **restart** and **start** commands are identical when arguments appear on the command line. Both reload the program being debugged, passing the specified arguments. However, they differ when they are used without arguments. The **restart** command with no arguments uses the same arguments used before; the **start** command with no arguments invokes the user application as if there were no arguments on its command line.

To stop **CPR** at any time, you need only issue the **quit** command (abbreviated as **q**). Keep in mind that you can restart your program without quitting **CPR** by means of the **restart** command (abbreviated as **res**). This command is discussed in greater detail in Section 7.

3.2 Preparing your Program for CodePRobe

The degree to which **CPR** can access symbols defined in any module is determined by the compiler debugging option used in compiling that module.

The **-d** option, when used by itself or immediately followed by a digit 1 through 5, activates the debugging mode of the Lattice C Compiler. When one of the debugging options is specified, the preprocessor symbol **DEBUG**

will be defined so any debugging statements in the source file will be compiled. (Note that the **-d** option has two uses, only one of which is concerned with saving information for the debugger.) Additionally, if the *lc* program is used to invoke the linker (using the **-L** option) and a debugging option is specified, the option ADDSYM is automatically passed to *Blink*. If you choose to call *blink* as a separate step, you must be sure to include the ADDSYM option in order to retain the external symbol information in the final executable file.

The following compiler debugging options are supported:

-d0 Disable all debugging information.

-d1 Output the line number/offset table only.

As a result of using the **-d1** option, the debugger can match the lines in a source file with the loaded code. This allows you to set breakpoints at source lines, and step through the source code. In addition, some limited symbolic information is available for external variables and functions. Only the address of a symbol is available. No information about the symbol's attributes is known. For this reason, the debugger will attempt to display any variable as an int by default with a warning message:

```
> d fahr
accessing extern with unknown attributes - assuming
int
1101004800
```

To display a variable in its proper form, use a type cast with the **display** command:

```
> d fahr (float)
20
```

or use one of the **dump** commands. (See Section 5 for details on the **display** and **dump** commands.) While the **-d1** option provides limited information to the debugger, it is useful with large programs or when disk space is limited because it produces much smaller object files. The

-d1 option also provides all the information needed by the disassembler, *omd*.

-d Same as **-d1**

-d2 Output full debugging information for those symbols referenced in the module.

The **-d2** option generates complete symbolic information for all symbols declared in a module. It permits symbolic access to automatic variables, formal parameters, register variables, all static symbols, bit fields, structure or union members, and enumeration constants. If a module is compiled with this option, you can set breakpoints at source lines, display or alter data symbolically, set watches on variables or areas of memory, and list source code. The **display** command will properly format all data declared in the module. It is important to realize that only symbol information for variables referenced in the module will be stored. If an external declaration, or structure tag definition is made, but no actual reference to the variable is generated in the code, the debugger will not be able to find any attribute information for that variable. Most source modules use numerous *include* files which contain external references that are never actually used. By limiting debug information to only include referenced symbols, the output module size is greatly reduced.

The compiler will often manipulate a variable's contents in registers over several lines of code. If a breakpoint is reached in this section of code, the debugger will display the value in memory, not in the register. If you suspect that the debugger is not displaying the proper values, the code can be inspected in mixed mode, or the file should be recompiled with the **-d3** option. Note that register variables, (i.e automatic variables that are only assigned to registers) will always be found in their proper register.

-d3 Output full debugging information for those symbols referenced in the module; also cause the code generator to flush all registers at line boundaries.

The **-d3** option generates exactly the same symbolic information as the

-d2 option. In addition, it causes the compiler's code generator to produce code to flush registers to memory after every C source line is executed. This can be useful because the debugger takes the value of a variable from its memory location, not a temporary register. While this option is recommended for use with the debugger, it causes inefficient code generation. Any module compiled with this option should be recompiled before it is placed into production.

-d4 Output full debugging information for all symbols declared in the module even if there is no reference to them in the generated code.

This option is similar to the **-d2** option except that information on all symbols is generated.

-d5 Output full debugging information for all symbols declared in the program even if there is no reference to them in the generated code; also cause the code generator to flush all registers at line boundaries.

This option is similar to the **-d3** option except that information on all symbols is generated.

3.3 Which Debugging Option To Use

In order to make full use of the debugger's features, you should always use either the **-d2**, **-d3**, **-d4**, or **-d5** option. If the compiler keeps the value of a variable in a register (as it may in order to eliminate unnecessary loading and storing of values in its registers), then the true value of a variable at any given time may be in a register rather than in the variable's memory location. As a consequence, the value of the variable displayed by the debugger may not be its correct value. The **-d3** and **-d5** options cause the correct values of variables to be in the memory locations of the variables at line boundaries, but at the cost of generating extra code required to store, and perhaps reload, the values.

Only the **-d3** and **-d5** options affect the quality of code generated by the compiler. The other options do not affect code generation and differ only in the amount of symbolic information they pass on to the linker. If you want to debug exactly the program you intend to use, you should use **-d2** or **-d4**

rather than **-d3** or **-d5**, and debug in mixed mode so that you can see the machine instructions corresponding to your C source lines. (See the description of the **set source** command later in this section.) When you do this, you will be able to tell if the value of a variable is being kept in a register rather than being flushed to memory, and you can then use the **register** command rather than the **display** or **dump** commands to determine the variable's value. (See Section 5 for more detail about these commands.)

If you are not concerned with debugging production quality code, you should use the **-d3** option, since the difference in generated code will have no effect. Even if you are attempting to create a piece of production software, you should begin your debugging by using the **-d3** option and then recompile with the **-d2** option when most of your bugs have been detected and repaired.

3.4 Determining Source File Location

CodePRobe knows where you compiled each C source module and will look in that directory first when invoked. However, you may wish to direct **CPR** to look for source files on a different disk or at different location on the same disk by using the **set search** command described later in this section.

If **CPR** cannot find the source file for the function it is currently executing, it displays the message:

```
Source not available
```

Your source code will be displayed as long as you have used some debugging option with the compiler (*except* the **-d0** option).

3.5 Choosing Command-Line Options

The **CodePRobe** environment can be customized by a number of available command-line options. A command-line option is included in the command line used to invoke the debugger in the following way:

cpr [*cpr options*] *application name* [*application arguments*]

As pointed out in Section 2, **CPR** is started by typing `cpr` at the command-line prompt and specifying the name of the program you want to debug. If there are any command-line options for the program you are debugging, you must include these on the command line as well. For example, the command:

```
cpr myprog myfile.txt 2
```

invokes **CPR** to debug the application program `myprog`, then passes `myprog` the command-line options `myfile.txt` and `2`.

You can use the following command-line options when invoking **CPR**.

Window Dimension options

Use the -wdialog, -wregister, -wsource, and -wwatch options to specify start-up window coordinates for the Dialog, Register, Source and Watch Windows, respectively. Window coordinates are measured in character positions. A width or height of 0 causes the window to extend to the screen border on the right or bottom, respectively. The option names can be abbreviated to two letters:

-wd[ialog]	*left*	*top*	*width*	*height*
-wr[egister]	*left*	*top*	*width*	*height*
-ws[ource]	*left*	*top*	*width*	*height*
-ww[atch]	*left*	*top*	*width*	*height*

For example:

```
cpr -ws 0 1 50 12 -wd 0 13 50 0 ftoc
```

Note that the Register and Watch Windows will not be displayed on startup; however, when they are opened by pressing the appropriate function key (either F1 or F4), they will open to the coordinates specified on the command line.

Workbench option

Use the **-w** option to setup **CPR** in the Workbench screen instead of a new screen. This option takes less system memory and is particularly useful when memory is in short supply.

Interlace option — Use the **-i** option to setup a screen in interlace mode. By default, **CPR** opens a new screen using the specifications set up by Preferences for Workbench screens. If you wish to set up a screen in interlace mode, include the **-i** option before typing the application command name.

Command option — Use the -command option (followed by specific commands) to execute debugger commands on startup. The commands are executed after the "go **main**" if the **-startup** option is not specified, or after the profile script if it is specified. (Profile scripts are discussed later in this section.) For example:

```
cpr -command "proceed; display fahr"
```

would execute to main, step over 1 line of code and display the variable `fahr` in the Dialog Window before giving control to the user.

Line option — Use the **-line** option to start **CPR** in line mode. Line mode causes the debugger to run in the current CLI window. The equivalent of the Dialog Window is provided, with no other windows available. Line mode is probably only useful for running from a script file, redirecting output using the CLI redirection facility, or running from a remote terminal over a communications port.

Startup option — The **-startup** option suppresses the automatic "go main." that is normally executed by the debugger on startup. This option is useful if your application redefines *main* or *c.a* so that no main function exists or if you wish to step through such code. If this option is used, no initialization of any kind will have been performed by the application process before control returns to the user.

If the **quit**, **start**, or **restart** commands are invoked and an application process has not exited, the debugger normally calls *exit* to cleanup any process resources that may not

have been freed. However, if the **-startup** option was used, *exit* will not be called.

Temporary file option

In order to access debugging information in an optimal manner, **CPR** creates a temporary file called a "spill file" to store information for quick access. Normally this file is located in **ram:**. The **-temp** *filename* option opens a temporary spill file called *filename* instead of the default *ram:cpr.tmp*. This option may be useful to move the spill file to some other location if your system has a limited amount of memory.

3.6 Setting and Showing Options Within CodePRobe

In addition to the command-line options discussed above, a number of options can be set from within **CPR** to customize the way information is displayed or handled. Most of these options are modified with the **set** command or through the Options menu. To see the default options while in the debugger, activate the Options menu or execute the **show** command. The following shows all of the default values.

```
> show
Source mode.........C
Autoswap mode.......On
Echo mode...........Off
Instruction bytes...On
Context.............2 lines
List default........6 lines
Unassemble default..4 lines
Display default.....%d
Search path:
```

3.6.1 Source Mode

The current source mode determines how **CodePRobe** responds to such commands as **trace** and **proceed**. These commands allow you to single step through your program and are discussed in more detail in Section 7. In

addition, the current source mode affects how information is displayed after a breakpoint and and how some commands format their output. In general, **CPR** uses the current source mode to decide whether to display a C source line, an assembly instruction, or both.

The current source mode can be determined by means of the **show** command or by activating the Options menu. It can be set or changed by means of the **set source** command or the Options menu.

3.6.2 C Mode

In C mode, **CodePRobe** displays C source lines when the **trace** or **proceed** commands are executed or when a breakpoint is triggered. To set the current source mode to C mode, enter the command:

```
> set source c
```

While in C source mode, you cannot single step by assembly instruction.

3.6.3 Assembly Mode

In assembly mode, diassembled code is displayed in the source window. Single step commands will step by assembly instruction.

To set the current source mode to assembly mode, use the command

```
> set source asm
```

3.6.4 Mixed Mode

In mixed mode, both assembly instructions and C source lines (if available) are displayed in the source window. For each C source line displayed, the corresponding assembly instructions are displayed as well.

To set the current source mode to mixed mode, give the command

```
> set source mixed
```

In mixed mode, you can take advantage of the two different forms of the **trace** and **proceed** commands in order to step by either C source line or assembly instruction.

3.6.5 Autoswap Mode

When in autoswap mode, the screen containing the application's output is pushed to the front each time **CodePRobe** gives control to the application. When a breakpoint is reached, the debugger screen is again pushed to the front. If you are single stepping through source code, the switching of screens will probably appear as a brief flash. If the autoswap mode is turned off, the application screen will not be pushed to the front. The autoswap feature can be turned on or off with the **set** command. The syntax is:

se[t] **a**[utoswap] [on | off]

Some examples:

```
> set auto on
> set a off
```

3.6.6 Echo Mode

When in echo mode, commands are echoed in the dialog window before being executed. This option is useful if you wish to execute debugger command files and see the command lines as they are executed, along with their output. (Command files will be discussed in the next section.) Commands that are invoked by menu selections or double-clicking the mouse are also displayed as they are executed. You will probably want to turn this feature off while entering commands in the Dialog Window because command lines are displayed as they are entered anyway. The syntax is:

se[t] **e**[cho] [on | off]

Some examples are:

```
> set echo on
> t
t
cat!main line 27
```

```
> set echo off
set echo off
> t
cat!main line 30
```

Notice that the **set echo off** command was echoed because the echo was not turned off until *after* the command was executed.

3.6.7 Instruction Bytes Option

The instruction bytes option affects the display format of disassembled code. It affects both the display in the Source Window and the output of the **unassemble** command (discussed in Section 9). If the option is on, the second field of the disassembly contains a hex dump of the instruction. The following fields contain the op-code and its operands. If the option is off, the hexadecimal dump is suppressed. The syntax is:

se[t] **i**[bytes] [on | off]

Some examples are:

```
> set ibytes on
> unassemble 7
main:
   7 {
0025F950 48E70130           MOVEM.L     D7/A2-A3,-(A7)
0025F954 BFEC0004           CMPA.L      0004(A4),A7
0025F958 65001BD6           BCS         00261530
> set i off
main:
   7 {
0025F950       MOVEM.L     D7/A2-A3,-(A7)
0025F954       CMPA.L      0004(A4),A7
0025F958       BCS         00261530
```

3.6.8 Context Lines Option

As you proceed through an application program by tracing and breakpointing, **CodePRobe** always keeps the current source line or assembler instruc-

tion displayed in the source window. **CPR** attempts to keep at least a minimum number of context lines of code displayed above and below the current line. The number of these context lines can be controlled by the context lines option. Note that, in mixed mode, it is not always possible to keep the right number of lines above or below, but the source code line will always be displayed and the assembler instruction will be displayed if possible. The default number of context lines is 2. The syntax is:

se[t] **c**[ontext] *number*

Some examples are:

```
> set context 4
> set c 2
```

3.6.9 List Default Option

The list default option determines the default number of source lines displayed by the **list** command. (The **list** command is discussed in Section 9.) The default value for this option is 6. The syntax is:

se[t] **l**[ist] *number*

3.6.10 Unassemble Default Option

The unassemble default option determines the default number of instructions displayed by the **unassemble** command. The default value for this option is 4. The syntax is:

se[t] **u**[nassemble] *number*

Note that the default value is only used when no source file is available. When a source file is available for a section of code, **unassemble** displays the disassembly for a single source line by default. (The **unassemble** command is discussed in greater detail in Section 9.)

3.6.11 Display Default Option

The display default option controls the format in which integers values are displayed. The choices are decimal and hexadecimal. By default decimal values are displayed. The syntax is:

se[t] **d**[isplay] [d | x]

Some examples:

```
> set display x
> d upper
0x12C
> set display d
> d upper
300
```

3.6.12 Setting the Search Path

The search path is used to find the source modules used when compiling the application. When a module is compiled with one of the debug options, the full path name of the source module is saved in the debug hunk. **CPR** always looks for the source module in its original location first, then it looks in the current directory, and finally, it walks through each of the directories specified in its search path in order. If you have moved the source module to some other location, use **set search** to add the new location to your search path. Note that you cannot change the actual name of the source file.

The **set search** command takes three forms:

se[t] **se**[arch] *dir* [, *dir* [...]]

se[t] **se**[arch] + *dir* [, *dir* [...]]

se[t] **se**[arch] - *dir* [, *dir* [...]]

The first form sets the search path to the list of directories on the command line. The second form appends the list of directories on the command line to the current search path. The last form deletes the list of directories from the current search path. Some examples are:

```
> set search c:, df0:, dh0:mysource
> set search + df1:test
> set search - df0:, c:
```

3.7 Profile Scripts

Since you will probably want **CodePRobe** to start the debugger with the same options most of the time, **CPR** allows you to create debugger command script files to be executed each time the debugger is invoked. The profile scripts are executed after the initial `go main`, but before any commands specified with the **-command** option are executed. Profile scripts must be named *profile.cpr*. **CPR** first looks for a script in the **s:** directory and, if successful, executes it; then **CPR** searches through the execution path defined by AmigaDOS. It will execute only the first script encountered in the path. (Note that the path always includes the current directory first and c: last).

You will probably want to set the options that you use for all projects in the *s:profile.cpr* script file, and place project-specific commands in a *profile.cpr* script file in the application's test directory. The following are examples of scripts displayed from the CLI window using the AmigaDOS command **type**:

```
> type s:profile.cpr
/* This is the debugger startup profile */
set search dh0:mysource, df0:mysource, ram:
set display h
set ibytes off
> type dh0:test/profile.cpr
/* This is the Lattice debugger profile */
/* script for a particular project      */
set source mixed
proceed
display argc, argv
```

Note that C style comments can be placed in profile scripts as long as they exist on a single line.

Profile scripts are discussed again in Section 9 when the **execute** command is introduced.

Section 4

Controlling CodePRobe

CodePRobe, like most other Amiga applications, makes extensive use of windows and pull-down menus. In addition, the function keys have a great deal of utility in controlling **CPR**.

Though many of the functions of **CPR** are available by using the mouse or pressing function keys, more sophisticated commands are available only by using the Dialog Window. A thorough knowledge of command-line editing can increase your proficiency using dialog commands.

Another means of controlling **CPR** is through use of the menu accelerator keys built into the program. By holding the right Amiga key in conjunction with selected keys, you can execute common commands with a few keystrokes.

This section explains how to get the most efficiency from **CodePRobe** by showing you alternative ways of issuing commands. In this respect, **CPR** gives you a great deal of flexibility, allowing you to work in the manner that best suits your needs.

4.1 Using Pull-Down Menus

The most common debugger activities can be performed using pull-down menus. As with other Amiga applications, you use the right mouse button to control menu functions. **CodePRobe** has five menus with sub-items as follows:

File	Options	Run	Break	Watch
Module	Source Mode	Trace	Set	Set
Line	AutoSwap Mode	Proc	List	Break
Execute	Echo Mode	Go	Clear	List
Refresh	Instruction Bytes	Once	Enable	Clear
Quit	Context Lines	Return	Disable	Enable
	List Default	Start	All Clear	Disable
	Unassemble Default	Where		All Clear
	Display Default			

4.2 Command-Line Editing

You will find that issuing commands in the Dialog Window offers much power and flexibility. For this reason, command-line editing techniques are well worth learning.

Editing functions on the numeric keypad are only available when the keypad is in the numeric mode. To toggle between the numeric and function modes, press F10.

The keypad functions are clearly marked on the numeric keypads of the Amiga 500 and 2000 models. However, the Amiga 1000 lacks these markings, so owners of this machine may want to take special note of the following functions:

Function	Numeric Keypad	Description
Ctrl X	-	erase the current line
Ctrl C	-	erase the current line
Ctrl End	Ctrl 1	erase from cursor to end of line
End	1	move cursor to end of line
Home	7	move cursor to start of line
Ins	0	start inserting at cursor
Ctrl A		toggle insert/overstrike
Del	decimal	delete character under cursor
←	4	move cursor left one position
→	6	move cursor right one position
Shift ←	Shift 4	move cursor left one word
Shift →	Shift 6	move cursor right one word
↵	ENTER	send line to program as input
↑	8	recall previous Dialog Window command
↓	2	next Dialog Window command
PgUp	9	page up in the Source or Dialog Window
PgDn	3	page down in the Source or Dialog Window

The current editing modes can be determined by examining the left end of the Dialog window's title bar. The first character will be an "I" if in insert mode or "O" if in overwrite mode. The second character is always a slash. The third character will be an "F" if the numeric keypad is in function mode or an "N" if in numeric mode.

4.3 Command Name Shortcuts

Command names need not be entered in their entirety. Generally, the minimum number of characters necessary to uniquely identify a command is re-

quired. The help menu shows required keys in caps for command names while the rest are shown in lower case. For example, RETurn in the help menu indicates that only the first three letters of the command are required, but typing in "return" or "retur" would also work.

In the section to come, topic headings for commands include the command abbreviations in parentheses following the command title. The syntax shows the optional portion of a command in square brackets [ ].

4.4 Special Function Keys

As a handy reference, the action of the function keys (F1 through F9) are described on the **CPR** screen title bar.

Function	Action
HELP	executes help command
F1	toggle for opening/closing Watch Window
Shift F1	toggle for opening/closing Source Window
F2	zoom the active window to the size of the screen
F3	recall last command
Shift F3	recall previous command
F4	toggle for opening/closing Register Window
F5	re-execute the last command
F6	activate the next window
F7	step into a line/instruction (trace)
F8	step over a line/instruction (proceed)
F9	toggle for swap screens between debugger and application
F10	toggle the numeric keypad
Ctrl L	refresh screen
Ctrl W	refresh screen

Note: An advantage of using special function keys and menu accelerators is

that they work regardless of which window is active. Dialog commands can only be entered when the Dialog Window is active.

4.5 Menu Accelerator Keys

Menu accelerator keys are familiar to most Amiga users. Menu items can be executed from the keyboard by striking the right-Amiga key in combination with another key. You can tell which menu options have accelerators by checking the menu display for the accelerator icon. Menu accelerator keys include the following:

Key	Menu	Option
A	Break	Clear all breaks
B	Break	Set a breakpoint
C	Break	Clear a breakpoint
D	Break	Disable a breakpoint
E	Break	Enable a breakpoint
F	File	Refresh screen
G	Run	Execute the go command
L	Break	List breakpoints
M	File	Change current module
O	Run	Once (go to a line or function)
P	Run	Proceed (step over the current line/instruction)
Q	File	Quit the debugger
R	Run	Return from current function
S	Run	Start (restarts the program with same command line)
T	Run	Trace (step into the current line/instruction)
W	Run	Where (display stack frame backtrace)
X	File	Execute a debugger command script from a file

4.6 Command Syntax

In order to understand the individual command descriptions, you must first understand the format of **CPR** commands, terminology used, and the command syntax associated with their use.

Each **CPR** command has a simple format:

```
command p1 p2 p3  . . .
```

where p1, p2, and p3 are parameters of the command. Some commands can be used with no parameters, some with one, and some require more than one parameter. In addition, some commands have alternate forms in which certain parameters are optional.

You can specify the command name in either uppercase or lowercase. Most commands possess an abbreviated version, but you can use the full command name if you want. When an abbreviated version is recognized, that part of the command name which is unnecessary is shown in square brackets.

If a parameter is optional, it will be enclosed in square brackets:

```
command [p1]
```

In some circumstances several alternative parameters are permitted. A vertical bar (|) separates permitted alternatives. Alternative parameters are enclosed in either square brackets or parentheses. (Neither brackets nor parentheses used for this purpose should be typed explicitly.) Square brackets indicate optional parameters.

For instance:

```
command ( p1 | p2 | p3 )
```

indicates that the command requires one parameter that can be either p1, p2, or p3, while the syntax:

```
command  p1 [ p2 | p3 ]
```

indicates that the command requires one parameter (p1) and has an optional second parameter that can be either p2 or p3.

Within a command, any symbol or expression that is enclosed in bold face is a literal part of the command. In particular, if a command must contain parentheses or brackets, you will notice that these are printed in bold face to distinguish these occurrences from their use in indicating optional parameters or alternatives. Most frequently you will notice that when a command is described, no specific parameters are mentioned; rather, the type of parameter (such as a *string* or *address*) is specified. Parameter types are always italicized. The parameter types required by **CPR** commands are described when the specific commands are introduced in later chapters.

For instance, the **break** command described in Section 7 has command syntax specified as follows:

> **b**[reak] *location* [*number*] [**when (** *condition* **)**] [*cmd_list*]

This means that the **break** command can be specified either as "break" or simply as **b**. There is one required parameter, *location*, as well as three optional parameters. The first optional parameter is a *number*. The second optional parameter is a "when" clause of the form:

> **when (** *condition* **)**

where the word **when** and the parentheses must be typed in as part of the command. Between the two parentheses, you must have a *condition* parameter; for instance, `i == 5`.

The third optional parameter is a *cmd_list*, which is basically a sequence of debugger commands that will be invoked at the point the *condition* set forth in the **when** clause is satisfied.

Thus, a specific example of a **break** command is

```
break main 9 when (i == 5)
```

which is of the form

> break *location* **when (** *condition* **)**

where the first optional parameter (*number*) and the second optional parameter (a "when" clause) are present. This command means: set a breakpoint at line 9 of the function main to be triggered when the variable `i` has the value 5.

Many commands require parameters that are machine addresses. In order to simplify the syntax of the commands and make their use easier, an automatic process of addressing will take place in many contexts. This means that in contexts that obviously require an address, **CodePRobe** will automatically convert the parameter you specify to the corresponding address.

For example, using the **da** command to dump a range of memory as ASCII characters, you may want to dump part of a character array as:

```
da &a[4] L 5
```

By referring to the addresses of the beginning and ending elements that you want displayed, this command will dump the fourth through eighth elements of the array. An equivalent command would be:

```
da a[4] L 5
```

where the operands will automatically be dereferenced by **CPR**.

Section 5

Examining Data and Data Structures

CodePRobe offers several commands which you can use to display the data referenced by your program. This section describes the following commands for examining data and data structures:

Display	Display an object or area of memory as a specific type of object.
Dump	Examine the value of any variable in a program, or examine an arbitrary range of memory.
Register	Display the current contents of the registers and the current flag settings, or alter the contents of registers.
Register Flag Dump and Set	Display the current settings of the status register flags, or either set or clear the current flag settings.
Whatis	Determine the type of a variable or the fundamental C type to which a typedef identifier resolves.

Before discussing these commands, let's discuss some of the command options associated with their use. This list is by no means comprehensive of all

the options used by **CPR** commands. Other parameters will be introduced along with the commands with which they are associated.

5.1 Command-Line Options

Depending on the command and its format, one or more of the following command-line options may be applicable.

5.1.1 Address

An *address* is any expression that denotes an address. This may be a constant, a C pointer variable, a register, or the result of prefixing "&" to a C identifier of the proper type. (Note that the address operator "&" may not be applied to the identifier of a bit-field.)

The following are examples of expressions that denote addresses:

```
&i
&array[3]
&mystruct-x
p
```

where `p` is a pointer and `i` is an int.

There are some cases where a register name may be ambiguous. For example, you may have defined a variable `sp` in your program. In the command:

```
db sp
```

it is not clear whether you mean to dump data beginning at the variable `sp`, or at the address contained in the register **sp**. In such cases **CPR** will assume that you mean to refer to the register **sp**. To refer to your variable `sp`, you can escape the register name with an "@" sign: `@sp`. Alternatively, you can specify the variable in **function\variable** form. Thus, both of the following:

```
@sp
main\sp
```

refer to a variable sp rather than to the register sp. The term sp will always be interpreted immediately as referring to the register.

In certain commands, it is necessary for **CodePRobe** to know the type of an object to be modified or displayed. If the object is referred to by means of an address constant or register, the value is assumed to be a character pointer.

5.1.2 Format

The *format* parameter refers to the form in which data is displayed. It may be either %d for decimal or %x for hexadecimal.

5.1.3 Number

A *number* is a number in decimal, octal, or hexadecimal notation. An initial 0 digit causes the number to be interpreted as octal, and an initial 0x (or 0X) causes the number to be interpreted as hexadecimal.

Examples of acceptable forms of the *number* parameter include:

```
12345
0455
0x380
0X1849
```

5.1.4 Range

A *range* is a contiguous area of memory that can be specified in one of two ways:

Form 1:

address address

Form 2:

address l *number*
address L *number*

In Form 1, the first *address* is the start address, and the range begins at this address.

In the first form, the second *address* is the end address, and the range includes the area of memory from the start address to the end address.

In the second form, the **l** or **L** indicates that *number* is to be interpreted as a length. In this case, *number* is a number of bytes and the range is from the start address through *address*+*number* - 1. When **CPR** sees a solitary **l** or **L** in a command (that is, one preceded and followed by a space), it is considered part of a *range* expression.

What if you have a variable l or L in your program and you want to refer to this variable in a **CPR** command? In this case, you must escape it with a @ or specify it in the **function\variable** form. Thus,

```
@l
@L
```

refer to the variables l and L, respectively, as do

```
main\l
main\L
foo\l
foo\L
```

The following fit the definition of the *range* parameter:

```
0x123456 0x123461
0x123456 L 11

a1 a2
a7 l 20

&p[0] &p[5]
p[0] p[5]

&p[0] l 6
p[0] l 6
```

The first pair of examples are equivalent. The next pair are variants of the basic *range* expression. The remaining pairs are equivalent ranges, given the addressing policy described earlier in this section.

5.1.5 Register

A *register* refers to the name of any of the 68000 registers. These include a0 through a7, d0 through d7, sp, and pc.

5.1.6 Typename

A *typename* can be any of the elementary data types provided by the C language including any of the following:

```
char
uns[igned] char
short
uns[igned] short
int
uns[igned] int
long
uns[igned] long
uns[igned]
float
double
```

A *typename* can also be any of the C types just listed followed by an asterisk (*).

In addition, the *typename* parameter can take any identifier defined by means of a typedef statement.

5.1.7 Variable

A *variable* is an expression in your program that designates an object. It can be either a simple identifier or an expression referring to an array element, structure member, or object pointed to by a pointer. In addition, it can begin with a specification of the form

module **!** *function* ****

where *function* is the name of any function in your program (including functions declared to be of the static storage class) and *module* is the root name of a file (i.e., that part of the file name exclusive of any path or extension).

The following are objects which fit the definition of *variable*:

```
i
max
array[3]
array
io!readfile\length
mystruct->name[2]
mystruct
*cptr
main!opnf\count
opnf\i
```

5.2 Display (d)

Among the most useful commands of **CodePRobe** is the **display** command. The **display** command displays C data structures and objects, and identifies an object or area of memory as a specified type of object.

The syntax of the **display** command is:

d[isplay] (*variable* | *address*) [**(** *typename* | *tag* **)**] [,] . . .

Note that the first set of parentheses are included only for clarity, while the second set should be typed in explicitly.

Either an *address* or a *variable* must be specified in the command. As mentioned earlier, The *typename* parameter can be any C type or any identifier defined by means of a typedef statement, and *tag* must be a structure or union tag defined in your program.

If an *address* parameter is used and that *address* is an address constant, then in the absence of a *typename* parameter, the *address* is assumed to be of type char *.

In addition to the basic format just given, the **display** command can accept multiple arguments, each of the form:

(*variable* | *address*) [(*typename* | *tag*)]

and separated by commas. Given this information, the following are examples of the **display** command:

```
> display i
> d 0xC80FF8
> d i (char)
> d cptr (struct date_time)
> d i, i (char), 0xC80FF8 (char *)
```

In looking at the examples to follow, assume that you have the following definitions in your source:

```
struct X { int a, b[3] , c; } x;
struct Y { int a, b; } y;
typedef struct X Y;
struct W {
  int x;
  struct Y y[2];
  int z;
  } w;

struct Z {
         char c;
         struct Y d;
         int e;
} z;

struct X3D {
  int a,
  b[2][3][2],      /* a major index, 24 bytes
                      apart, next 8, minor 4 */
  c; }
  x3d;

int i;
int b[10];
```

Displaying the value of an integer variable is a simple matter:

```
> d i
3
```

To display its address, use the following command:

```
> d &i
0xC309B8
```

It is equally simple to display an array element:

```
> d *b
5
> d b[0]
5
```

or an entire array:

```
> d b
{
 [0] = 5
 [1] = 15
 [2] = 25
 [3] = 35
 [4] = 45
 [5] = 55
 [6] = 65
 [7] = 75
 [8] = 85
 [9] = 95
}
```

Structures are also easy to analyze using the **display** command:

```
> d x
struct X {
   a = 1
   b = {
        [0] = 2
        [1] = 3
```

```
            [2] = 4
            }
        c = 5
        }
> d z
struct Z {
    c = 20
    d = struct Y {
        a = 21
        b = 22
        }
    e = 23
}
```

Multi-dimensional arrays are displayed in a format which enhances analysis:

```
>  d x3d
struct X3D {
    a = 40
    b = {
        [0] = {
            [0] = {
                [0] = 41
                [1] = 42
              }
            [1] = {
                [0] = 43
                [1] = 44
              }
            [2] = {
                [0] = 45
                [1] = 46
              }
          }

        [1] = {
            [0] = {
                [0] = 47
                [1] = 48
              }
            [1] = {
                [0] = 49
                [1] = 50
```

```
                }
              [2] = {
                  [0] = 51
                  [1] = 52
                }
            }
        }
    c = 53
}
```

Normally the object specified in the **display** command is displayed in its "natural" format. However, by using the *typedef* or *tag* option of the command, you can display an object of one type as though it were an object of another type.

For example, you can first dump a structure in its "natural" form:

```
> d z
struct Z {
   c = 20
   d = struct Y {
      a = 21
      b = 22
      }
   e = 23
}
```

Then you can dump it as though it were really a different sort of structure:

```
> d z (struct W)
struct W {
   x = 20
   y = {
        [0] = struct Y {
          a = 21
          b = 22
          }

        [1] = struct Y {
          a = 23
          b = 30
          }
```

```
        }
    z = 31
    }
```

This facility (known as "casting") may be useful only for somewhat odd circumstances in C. However, you may find it most useful in dumping a memory object defined in assembly language as though it were a C structure. You can display an arbitrary area of memory as any type of object defined either by the C language or within your program, as in the following example:

```
> d 0xc85400 (struct W)
struct W {
   x = -166
   y = {
       [0] = struct Y {
         a = 18059
         b = -31991
         }
       [1] = struct Y {
         a = 3524
         b = -13475
         }
      }
   z = 26740
   }
> d 0xc85400 (X)
struct Y {
   a = -166
   b = 18059
   }
> d 0xc85400 (struct X)
struct X {
   a = -166
   b = {
       [0] = 18059
       [1] = -31991
       [2] = 3524
      }
   c = -13475
}
```

Whether this display is meaningful depends of course upon what data you have stored at that location. Notice that if the term `struct` is omitted from the *tag*, the debugger looks first for a typedef symbol and then looks for a structure, union, or enumeration tag only if such a typedef symbol has not been found.

In a similar fashion, you can display an arbitrary area of memory as though it were one of the fundamental C types. For example, an area can be displayed as though it were a double, a long, or a character pointer:

```
> d 0xC80000 (double)
2.40325e-242
> d 0xC80000 (long)
1183579994
> d 0xC80000 (char *)
0xC8FF5A
```

This facility may be useful if you are constructing one type of object in an area of memory that you have defined to be of another type (as in constructing a double as a sequence of bytes) or if you need to look at memory defined in an assembly module.

As mentioned earlier, the **display** command is capable of accepting several parameters separated by commas. You may find this particularly useful within breakpoint command lists to format your displays. A simple example involves displaying the value of `a[3][3]` and labeling as follows:

```
> d "a[",i,"]","[",i,"]  = ", a[i][i]
a[3][3] = 337
```

Here the string literals:

```
"a["        "]"         "["          "] ="
```

are specified by enclosing them in double quotes, while the values of `i` and `a[i][i]` are specified in the usual way.

5.3 Dump Commands

A **dump** command provides a simple way for you to examine the value of any variable in your program or to examine an arbitrary range of memory.

There are, in fact, several different **dump** commands, and each one is devoted to displaying an area of memory as a particular type of object or as a sequence of a particular type of object. You can think of the **dump** commands as more primitive and shorthand forms of the **display** command. Each **dump** command is of the format:

d? (*variable* | *address* | *range*)

where **?** is one of the following letters: **a, b, c, d, f, i, l, p, s, w,** or **z**. The dump commands correspond to the following actions:

Dump Command	Dumps as . . .
da[scii]	ASCII characters
db[yte]	bytes
dc[har]	characters
dd[ouble]	doubles
df[loat]	floats
di[nteger]	integers
dl[ong]	longs (same as **di**
dp[ointer]	a hexadecimal address
ds[hort]	short integers (same as **di**)
dw[ord]	2-byte words in hex
dz[ero]	null-terminated ASCII string

Some **dump** commands permit an optional format parameter which allow you to select the format in which the data will be displayed. These commands take the form:

d? (*variable* | *address* | *range*) *format* **-text**

where *format* is either **%d** or **%x.** If the **-text** option is specified, ASCII characters corresponding to the numeric values are displayed.

In the examples which follow, a number of variables are used as operands to the **dump** and **display** commands. You can assume that these variables have been declared as follows:

```
char chr;
char *stringptr;
int i;
int *intptr;
short sht;
short *shortptr;
long lng;
long *longptr;
float flt;
float *floatptr;
double dbl;
double *doubleptr;
```

If an *address* parameter is specified, memory is dumped beginning at that address. If an *address* is not specified, a process of addressing takes place in evaluating the *variable* parameter. If a *variable* is specified and it is not an *address* (that is, if it is not a pointer, address expression, or array name), then memory is dumped beginning at the address of *variable*.

For example, the following commands:

```
> di intptr
> di *intptr
```

are equivalent; both mean "dump the area of memory pointed to by `intptr`."

On the other hand, to dump the value (contents) of `intptr`, use the command:

```
> di &intptr
```

A **dump** command can be used to dump the memory at the address of a variable. For instance, the following command:

```
> di i
```

causes memory to be dumped starting at the address of `i`. (That is, the value of the variable `i` will be dumped.)

Since the variable in this case (`i`) is not an *address*, this command is equivalent to:

```
> di &i
```

Both commands mean "dump the value of `i`" or "dump what is at `i`'s position in memory."

In general, when a *variable* parameter is specified, the amount of memory dumped is determined by the size of the object (or in the case of a pointer, the size of the object to which it points). However, in the case of a character pointer or an address constant, 64 bytes of data are always displayed.

When a *range* parameter is specified, exactly that amount of data is displayed, provided that there is an integral number of the specified type of data items in the given range. If there is not an integral number of data items in that range, **CodePRobe** will "round down" the range to ensure that an integral number of the data items can be displayed. This may mean that a certain portion of the range you have explicitly specified is not displayed because it forms only a part of the type of object you are asking to display. For example, if you use the command:

```
> dd a7 l 20
```

to display 8-byte doubles, **CPR** will take this to be equivalent to:

```
> dd a7 l 16
```

The result of either command is that only two doubles may be displayed.

When displaying a range of data or a number of data items, the display may occupy several lines of screen space. In such circumstances the size of the lines in the display is determined by the width of the window in which they

are displayed. If you are debugging in line mode rather than window mode, a default line size is in effect.

5.3.1 Dump in ASCII Format (da)

The **da** command is used to display a block of data in ASCII character format. Normal printable characters appear as expected, and certain non-graphic characters appear as their usual C escape sequences:

```
\b      backspace
\f      form feed
\n      new line
\r      carriage return
\t      horizontal tab
\v      vertical tab
\0      null byte (string terminator)
```

Other characters are represented by the two-digit hexadecimal codes.

For example:

```
> da stringptr l 128

C80300   C  P  R  O  B  E     s  y  m  b  o  l  i  c
C80310   d  e  b  u  g  g  e  r \n  C  o  p  y  r  i  g
C80320   h  t \t  L  a  t  t  i  c  e     I  n  c  .  ,
C80330      1  9  8  8 \n  T  h  i  s     i  s     a
C80330   n  u  l  l     t  e  r  m  i  n  a  t  e  d
C80340   s  t  r  i  n  g     w  i  t  h     a     b  e
C80350   l  l 07     c  h  a  r  a  c  t  e  r     i  n
C80360      i  t  . \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
```

5.3.2 Dump as Bytes in Hex and ASCII Format (db)

The **db** command provides you with a way of displaying both the hex values of bytes and their ASCII representations. Its output is similar to that of the **da** command except that the display of each line is separated into a hex portion (to the left) and an ASCII portion (to the right). In the ASCII dis-

play portion, the high-order bit of each byte is ignored (that is, this bit is masked off prior to displaying the character). Any bytes that would not be displayed as standard printable characters are represented as periods (.).

Using the same example as in the **da** command yields the following results:

```
> db stringptr l 128

C80300  43 50 52 4F 42 45 20 73 79 6D 62 6F 6C 69 63 20  CPROBE symbolic
C80310  64 65 62 75 67 67 65 72 0A 43 6F 70 79 72 69 67  debugger.Copyrig
C80320  68 74 09 4C 61 74 74 69 63 65 20 49 6E 63 2E 2C  ht.Lattice Inc.,
C80330  20 31 39 38 38 0A 54 68 69 73 20 69 73 20 61 20   1988.This is a
C80330  6E 75 6C 6C 20 74 65 72 6D 69 6E 61 74 65 64 20  null terminated
C80340  73 74 72 69 6E 67 20 77 69 74 68 20 61 20 62 65  string with a be
C80350  6C 6C 07 20 63 68 61 72 61 63 74 65 72 20 69 6E  ll..character in
C80360  20 69 74 2E 00 00 00 00 00 00 00 00 00 00 00 00   it.............
```

5.3.3 Dump in Character Format (dc)

The **dc** command is like the **db** command except that it displays its values in decimal format rather than hexadecimal format. In addition, without the **-text** option, the **dc** command does not display ASCII character equivalents to the right of the numeric values. For example, compare the results of these two commands when applied to the same area of memory:

```
> db intptr
C80B98  05 01 04 00 03 00 07 00 11 00 1B 00 25 00 2F 00  ............/...
C80BA8  39 00 6B 00 75 00 7F 00 89 00 93 00 9D 00 CF 00  9.k.u...........
C80BB8  D9 00 E3 00 ED 00 F7 00 01 01 33 01 3D 01 47 01  ..........3.=.G.
C80BC8  51 01 5B 01 65 01 97 01 A1 01 AB 01 B5 01 BF 01  Q. .e...........
>dc intptr
C80B98     5    1    4    0    3    0    7    0
C80BA0    17    0   27    0   37    0   47    0
C80BA8    57    0  107    0  117    0  127    0
C80BB0  -119    0 -109    0  -99    0  -49    0
C80BB8   -39    0  -29    0  -19    0   -9    0
C80BC0     1    1   51    1   61    1   71    1
C80BC8    81    1   91    1  101    1 -105    1
C80BD0   -95    1  -85    1  -75    1  -65    1
```

Notice that since no length was specified in these cases, the default of 64 bytes takes effect.

5.3.4 Dump in Double Format (dd)

The **dd** command causes data to be dumped as 8-byte floating point numbers in IEEE format. The result is similar to calling the C *printf* function with a "%g" format.

For example, compare the results of two variations of the **dd** command:

```
> dd a7 l 32
C80144  1.23457E+12 8.45922E+28
C80154  2.58945E+10 4.82930E+22

> dd dbl
C80144  1.23457E+12
```

5.3.5 Dump in Integer or Word Format (di, ds, and dw)

The **di** command causes data to be dumped in long (or 32-bit) integer format, using decimal representations of the integers. Note that the **di** command cannot distinguish between modules compiled with 16-bit integers (using the -w option) or those compiled with 32-bit integers (using the default).

The **ds** command causes data to be dumped in short (or 16-bit) integer format, using decimal representations of the integers.

The **dw** command also dumps data in short (16-bit) format. For example, compare the results of the **ds** and the **dw** commands:

```
> ds i
C8000C 16

> dw i
C8000C 10
```

The same display produced by the **dw** command would result from the command:

```
> ds i %x
```

Dumping part of the area of memory used earlier to illustrate the **da** and **db** commands would yield:

```
> di 0xC80300 l 32
C80300  1129337423 1111826547 2037211759 1818845984
C80310  1684365941 1734829426  172191600 2037541223
```

```
> dw 0xC80300
C80300  4350 524F 4245 2073 796D 626F 6C69 6320
C80310  6465 6275 6767 6572 0A43 6F70 7972 6967

> di 0xC80300 l 32 %x
C80300  4350524F 42452073 796D626F 6C696320
C80310  64656275 67676572 0A436F70 79726967
```

5.3.6 Dump as Floats (df)

The **df** command dumps an area of data as though it contains single-precision floating point numbers. The display is similar to that produced by the **dd** command, but the data dumped are assumed to be in the 4-byte IEEE short floating point format. By default, the format used to display the data is similar to the "%g" format of *printf*. That is, each float is dumped either in "%f" or "%e" format depending upon which is most appropriate. To force the **df** command to dump the data in one of these formats, use either the "%f" or "%e" option.

If dumped as floats, the area of memory used in the previous examples appears as follows:

```
> df 0xC80300 l 32
C80300  +208.3215179   +49.2816887    +7            +1
C80310  +1             +1             +             +7
```

5.3.7 Dump in Long Format (dl)

The **dl** command causes the data to be dumped as long (32-bit) integers.

The **dl** command is provided as an alias for the **di** command a convenience to you in case you are accustomed to using a debugger with such a command.

5.3.8 Dump In Pointer Format (dp)

To see a range of memory dumped as a series of pointers, you can use the **dp** command. This is equivalent to specifying the %x option to the **dl** command:

```
> dp stringptr l 32
C80208   00C84568 00C84588 00C84320 00C62380
C80218   00C62888 00C43218 00C43220 00000000
```

5.3.9 Dump as Null-Terminated ASCII String (dz)

The **dz** command is used to display an ASCII string terminated by a null ("\0") character. To display a string (that is, a null-terminated sequence of characters) use the **dz** command:

```
> dz stringptr
"this is a string"
```

The string at that location is displayed in double quotes. Normally you would use either a character pointer or an array name as the argument to this command.

5.4 Register (r)

The **register** command has two forms, with and without parameters:

r[egister]

r[egister] *register* [=] (*variable* | *number* | *address*)

The first form of the **register** command displays the contents of all the registers, while the second form results in the specified value being placed into the named register.

5.4.1 Displaying the Contents of Registers

If you use the **register** command without parameters, the current contents of the registers and the current flag settings are displayed:

```
> r
d0=00242F54  d1=00242F54  d2=000003ED  d3=00000000  d4=00000000
d5=00000000  d6=00000000  d7=00000001  a0=00000000  a1=00241388
a2=00242F54  a3=0022aC36  a4=00221810  a5=00241A8E  a6=00203308
sp=00241A4A  ss=00000000  pc=0023C064  nt us di na pl nz nv nc
cat!filecopy line 10 - entry
```

In addition, the current address of the executing program is displayed.

5.4.2 Altering the Contents of Registers

The second form of the **register** command is similar to the **enter** command. It allows you to alter the contents of registers. You can specify the value you want placed in a register by means of a number, an address, or a variable in your program. For example:

```
> r
d0=00242F54  d1=00242F54  d2=000003ED  d3=00000000  d4=00000000
d5=00000000  d6=00000000  d7=00000001  a0=00000000  a1=00241388
a2=00242F54  a3=0022aC36  a4=00221810  a5=00241A8E  a6=00203308
sp=00241A4A  ss=00000000  pc=0023C064  nt us di na pl nz nv nc
cat!filecopy line 10 - entry
> r d7 5
> r a2 0x241a4a
> r
d0=00242F54  d1=00242F54  d2=000003ED  d3=00000000  d4=00000000
d5=00000000  d6=00000000  d7=00000005  a0=00000000  a1=00241388
a2=00241A4A  a3=0022aC36  a4=00221810  a5=00241A8E  a6=00203308
sp=00241A4A  ss=00000000  pc=0023C064  nt us di na pl nz nv nc
cat!filecopy line 10 - entry
```

Note that the current source line is displayed immediately after the display of the registers and flags.

You can also set a register by using a program variable:

```
> d i
12
> r d3 i
> r
d0=00242F54  d1=00242F54  d2=000003ED  d3=0000000C  d4=00000000
d5=00000000  d6=00000000  d7=00000005  a0=00000000  a1=00241388
a2=00241A4A  a3=0022aC36  a4=00221810  a5=00241A8E  a6=00203308
sp=00241A4A  ss=00000000  pc=0023C064  nt us di na pl nz nv nc
cat!filecopy line 10 - entry
```

5.5 Register Flag Dump and Set (rf)

The **rf** command allows you to display, set, or clear the current settings of the various flags. The syntax is:

> **rf** [*flag setting* . . .]

Used without parameters, it is equivalent to the **register** command. With one or more arguments, the indicated flags are set or cleared according to the flag settings specified.

A *flag setting* can be one of the terms in the following table. The Set column comprises the terms used to set flags, while the Clear column comprises the terms used to clear flags.

Flag Name	SET	CLEAR
Overflow (yes/no)	OV	NV
Interrupt (enable/disable)	EI	DI
Sign (negative/positive)	NG	PL
Zero (yes/no)	ZR	NZ
Auxiliary Carry (yes/no)	AC	NA
Carry (yes/no)	CY	NC
Trace (enable/disable)	TR	NT
State (user/supervisor)	US	SS

Flag settings can be specified in either uppercase or lowercase. Thus, both "CY" and "cy" indicate that the carry flag is set.

To set flags to certain values, use the **rf** command with the desired flag settings as follows:

```
> rf
d0=00242F54  d1=00242F54  d2=000003ED  d3=0000000C  d4=00000000
d5=00000000  d6=00000000  d7=00000005  a0=00000000  a1=00241388
a2=00241A4A  a3=0022aC36  a4=00221810  a5=00241A8E  a6=00203308
sp=00241A4A  ss=00000000  pc=0023C064  nt us di na pl nz nv nc
cat!filecopy line 10 - entry
> rf ov zr cy
> rf
d0=00242F54  d1=00242F54  d2=000003ED  d3=0000000C  d4=00000000
d5=00000000  d6=00000000  d7=00000005  a0=00000000  a1=00241388
a2=00241A4A  a3=0022aC36  a4=00221810  a5=00241A8E  a6=00203308
sp=00241A4A  ss=00000000  pc=0023C064  nt us di na pl zr ov cy
cat!filecopy line 10 - entry
```

If you specify inconsistent flag settings, the last of these will be used to set the flag in question:

```
> rf ei di nz
> rf
d0=00242F54  d1=00242F54  d2=000003ED  d3=0000000C  d4=00000000
d5=00000000  d6=00000000  d7=00000005  a0=00000000  a1=00241388
a2=00241A4A  a3=0022aC36  a4=00221810  a5=00241A8E  a6=00203308
sp=00241A4A  ss=00000000  pc=0023C064  nt us di na pl zr ov cy
cat!filecopy line 10 - entry
```

Here the `ei` setting is overridden by the later `di` setting.

5.6 Whatis (wha)

The **whatis** command is used to determine the type of an object or the configuration of a type of object. It has two forms:

> whatis *variable*
> whatis ((*typename* | *tag*))

where *typename* and *tag* are as described in the **display** command.

Assume you have defined the following objects:

```
struct X (
   int a;
   char b[3] ;
   double d;
} x;

typedef struct X *Y;
```

then the **whatis** command yields

```
> whatis x
extern at 00221FBC (16 bytes)
struct X
> whatis x.a
at 00221FBC (4 bytes)
long
> whatis x.b[0]
at 00221FC0 (1 bytes)
char
> whatis (Y)
struct X *
> whatis (int)
signed 32-bit integer
> whatis (struct X)
struct X ( (16 bytes)
   long a;
   char b[3];
   double d;
```

```
    };
```

Notice that the typedefs are expanded in terms of the more fundamental types.

Section 6

Modifying Code or Data

This section describes the commands you can use to modify data or code in your program. Commands described here include:

Enter Modify memory by either assigning a value to a variable or replacing data at a specified address.

Fill Fill a specified range or area of memory with the values provided.

Memcpy Copy the contents of one address to the contents of another address.

Strcpy Initialize strings or character arrays, or copy strings from one area of memory to another.

If you are debugging at the C source level, you are most likely to make use of the **enter** and **strcpy** commands. The **fill** and **memcpy** commands will be of more interest to you if you are debugging at the assembly level or need to initialize a large C structure or array.

Each of these commands allows you to alter memory directly, so they should be used with some caution.

In order to ensure that you achieve the expected results using these commands, you should compile your C source modules with the **-d3** or **-d3** option. This will guarantee that:

- no values of C variables are kept in registers without their memory locations being updated,
- and the compiler regards what is currently in a variable's memory location as that variable's value.

Without this guarantee, it is possible that the true value of a variable is stored in a register and not yet reflected in the variable's memory location. As a result, what the debugger considers the current value of the variable may not be its true value, and your debugging process could be hampered by this confusion.

In order to illustrate the data modification commands, we will use the same data definitions introduced in the previous section. In addition, assume that the following data items have been defined:

```
int im;
short jm;
char *cp;
char carr[100];
```

The following discussion uses the data definitions used to explain the **dump** commands in the previous chapter.

6.1 Enter (e)

The **enter** command is used to modify memory. This command takes three different forms:

e[nter] *variable* [=] (*variable* | *number* | *address*)

e[nter] *address* (*ctype*) [=] *list of values*

e[nter] *address* [=] *string*

The first form assigns the value of a variable in your program. The second

form initializes or resets the values in an array. The third form copies a string to a specified address without the null character (\0) which typically terminates strings.

6.1.1 Changing the Value of a Variable

You are most likely to use the first form of the **enter** command in debugging C programs. It allows you to change the value of a variable in your program by referring to that variable by name. In this context, *variable* has a rather broad sense that includes references to array elements, structure members, and indirect references.

The first form of the **enter** command uses the following syntax:

> **e**[nter] *variable* [=] (*variable* | *number* | *address*)

The variables used in this command must be of some scalar type. In general, neither of the two variables can refer to an aggregate such as a structure, union, or array. (The single exception to this is that the second variable can be an array name, in which case it is treated as the address of the array.)

If necessary, the *variable* or *number* being assigned should be converted to the proper type in accordance with the normal C conversion rules; it is then entered into memory at the address of the first variable operand. This is similar to a C assignment statement for a scalar variable.

Setting a simple variable to a new value is easy. Say, for example, you want to change the value of the variable `im` from -2 to -4:

```
> d im
-2
> e im -4
> d im
-4
```

It is just as easy to set one variable to the value of another. In the following case, the value of `im` is set to the value of `jm`:

```
> d jm
-2
> e im = jm
```

```
> d im
-3
```

(Notice that the "=" sign is optional in the **enter** command.)

Fields in structures can be set in a similar way. For example, you may want to give **CodePRobe** commands that would accomplish the same actions as the C statements:

```
w.x = 1;
w.z = jm;
w.y[0].a = w.y[1].b;
```

To accomplish this, you can use the **enter** command as follows:

```
> d w
struct W {
   x = 0
   y = {
       [0] = struct Y {
         a = 256
         b = -29436
         }
       [1] = struct Y {
         a = 1
         b = 6400
         }
      }
   z = -29436
   }
> e w.x = 1
> e w.z = jm
> e w.y[0].a = w.y[1].b
> d w
struct W {
   x = 1
   y = {
       [0] = struct Y {
         a = 6400
         b = -29436
         }
       [1] = struct Y {
```

```
            a = 1
            b = 6400
            }
         }
      z = -3
      }
```

(The **display** commands used in the previous example are not necessary, of course, but are included to illustrate how the values of the variables change.)

Pointers can also be given new values. For instance, the address of the structure `y` can be assigned to the character pointer `cp`. After this assignment you can see that the value of `cp` is the same as the address of `y` and that if the area of memory pointed to by `cp` is displayed as a structure of type `Y`, it is the same as the structure `y` itself.

```
> d cp
C83748
> d &y
C80C56
> e cp &y
> d cp
C80C56
> d y
struct Y {
   a = 1
   b = -123
   }
> d *cp (struct Y)
struct Y {
   a = 1
   b = -123
   }
```

6.1.2 Initializing or Resetting the Values in an Array

The second form of the **enter** command is useful when you wish to initialize or reset the values in an array. The **enter** command allows you to replace data at a specified address.

The second form of the **enter** command uses the following syntax:

> **e**[nter] *address* **(** *ctype* **)** [=] *list of values*

The *list of values* parameter is a sequence of elements, each of which is either a *variable* or *number*. Each element in the list is separated from the next by a space. Once the **enter** command is executed, each element in the list is converted to *ctype* and entered into memory in sequence beginning at *address*.

A simple example involves changing the first eight elements of the array `b`:

```
> d b
   {
    [0] = 5
    [1] = 15
    [2] = 25
    [3] = 35
    [4] = 45
    [5] = 55
    [6] = 65
    [7] = 75
    [8] = 85
    [9] = 95
   }
> e b (int) 2 4 6 8 10 0xFF 0xEE 0xDD
> d b
   {
    [0] = 2
    [1] = 4
    [2] = 6
    [3] = 8
    [4] = 10
    [5] = 255
    [6] = 238
    [7] = 221
    [8] = 85
    [9] = 95
   }
```

The *ctype* parameter indicates how the data following it are to be interpreted. The rather odd-looking command

```
> e b (char) 'a' 'b' 'c' 'd'
```

yields the equally odd-looking result

```
>d b
   {
   [0] = 1633837924
   [1] = 4
   [2] = 6
   [3] = 8
   [4] = 10
   [5] = 255
   [6] = 238
   [7] = 221
   [8] = 85
   [9] = 95
   }
```

The result becomes clearer when you display the data in `b` in another format:

```
> da b
00C80BAA:    a  b  c  d \0 \0 \0 04 \0 \0 \0 06 \0 \0 \0 \b
00C80BBA:   \0 \0 \0 \n \0 \0 \0 FF \0 \0 \0 EE \0 \0 \0 DD
00C80BCA:   \0 \0 \0  U \0 \0 \0 _
```

It is clear that the debugger performed the command and filled the initial portion of the integer array with the specified characters.

By using the *ctype* parameter in the **enter** command, you can fill an arbitrary area of memory with a sequence of items of a given type.

6.1.3 Copying a String to a Specified Address

The third form of the **enter** command copies a sequence of printable characters to a variable or a memory address without terminating the string with a null character. In contrast, the **strcpy** command (to be presented later in this section) *does* copy an implicit null character (\0) at the end of the string.

The third form of the **enter** command uses the following syntax:

e[nter] *address* [=] *string*

where *string* is a doubly quoted sequence of ASCII characters in the manner of C. If you want to include a double quote in the string, it must be escaped with a backslash (\).

This form of the **enter** command allows you to choose a particular address and enter a string into it. Selecting an arbitrary address could yield the following results:

```
> da 0xc80300
C80300   a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  p
C80310   a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  p
C80320   a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  p
C80330   a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  p
> e 0x0c80300 "this is a string"
> da 0x0c80300
C80300   t  h  i  s     i  s     a     s  t  r  i  n  g
C80310   a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  p
C80320   a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  p
C80330   a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  p
```

Notice that there is no null character after the string. To ensure that the string is null terminated, you can include a null character in your string explicitly:

```
> e 0x0c80300 "this one is null terminated\0"
> da 0x0c80300
C80300   t  h  i  s     i  s     a     s  t  r  i  n  g
C80310  \0  b  c  d  e  f  g  h  i  j  k  l  m  n  o  p
C80320   a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  p
C80330   a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  p
```

6.2 Fill (f)

The **fill** command copies particular values to a specified range of memory. It is designed to handle those instances in which you want to fill an area of memory with a value or list of values. In some ways, the **fill** command is similar to the **enter** command; however while the **enter** command takes as a

parameter either a variable or an address, the **fill** command takes as a parameter a *range of memory*.

The **fill** command takes either of two forms:

> **f**[ill] *range* (*ctype*) *list of values*
>
> **f**[ill] *range string*

The first form of the **fill** command involves a list of values, while the second form involves a given string (i.e., a single value). The *list of values* is identical to the same parameter in the **enter** command, and the parentheses are typed in as shown.

6.2.1 Filling Memory with a List of Values

In its first form, the **fill** command copies the *list of values* is copied iteratively and repetitively into memory beginning at the start address specified in *range*.

If the *list of values* is not large enough to fill the *range*, it will be copied iteratively into the *range* until the end of the *range* is reached. If the *range* is too small to accommodate the *list of values*, the *list of values* is truncated to the size of the *range*.

To clear the 40-byte (10-integer) array `b`, you could use the **fill** command in its first form as:

```
> f b[0] l 20 (int) 0
> d b
   {
    [0] = 0
    [1] = 0
    [2] = 0
    [3] = 0
    [4] = 0
    [5] = 0
    [6] = 0
    [7] = 0
    [8] = 0
    [9] = 0
   }
```

6.2.2 Filling Memory with a String

In the second form of the **fill** command, the given string is copied into memory beginning at the start address of *range*. The terminating \0 is not copied. If you wish to ensure that the resulting string is null terminated, include an explicit \0 before the terminating double quote. If the end address of *range* is reached before the end of *string* or before the end of an instance of the *list of values*, the remaining data (in the *string* or *list of values*) are not entered into memory.

Suppose you wish to copy the string `"ab"` to memory starting at `b` and filling 20 bytes:

```
> f b l 20 "ab"
> da b l 20
C80BAA   a  b  a  b  a  b  a  b  a  b  a  b  a  b  a  b
C80BBA   a  b  a  b
```

Notice that in this case, the *range* size exceeds the size of the string, so the string is repeatedly copied into the *range*.

If the string used to fill the *range* is too long, it will be truncated to the appropriate size:

```
> f b l 10 "this string is too long to fit"
> da b l 20
C80BAA   t  h  i  s     s  t  r  i  n  a  b  a  b  a  b
C80BBA   a  b  a  b
```

6.3 Memcpy (mem)

The **memcpy** command is similar to the C language library function of the same name. It copies the contents of one address to the contents of another address. Its syntax is:

mem[cpy] *address address number*

where the first *address* is the destination and the second *address* is the source. The *number* is the number of bytes to be copied. The **memcpy** function copies *number* bytes of data to its first operand (destination) address from its second operand (source) address.

You can copy the contents of the array `b` into the area of memory pointed to by the character pointer `cp` as follows:

```
> di b l 20
C80BAA  1952999795  543781664  1629516660  1919512167
C80BBA     6644341
> di cp l 20
2482E5           0          0           0           0
2482F5           0
> memcpy cp b 20
> di cp l 20
2482E5  1952999795  543781664  1629516660  1919512167
2482F5     6644341
```

Any addresses in memory can be specified for the destination and source.

6.4 Strcpy (str)

The **strcpy** command corresponds to the C library function of the same name. It copies a null-terminated source string to the destination area. It is similar to **memcpy** in syntax:

> **str**[cpy] *address* (*address* | *string*)

The first *address* operand is the destination, and the second operand is either a source address or a source string. The null character (\0) terminating the string is copied also. It is assumed that the second operand points to a null-terminated ASCII string. Violation of this assumption yields unexpected results.

You can explicitly copy a string constant into an array. For instance

```
> da carr
C40B26  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
C40B36  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
C40B46  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
```

```
C40B56  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
C40B66  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
C40B76  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
C40B86  FF FF FF FF
> strcpy carr[20] "this string will be null terminated"
> da carr
C40B26  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
C40B36  FF FF FF FF  t  h  i  s     s  t  r  i  n  g
C40B46   w  i  l  l     b  e     n  u  l  l     t  e  r
C40B56   m  i  n  a  t  e  d \0 FF FF FF FF FF FF FF FF
C40B66  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
C40B76  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
C40B86  FF FF FF FF
```

copies the string "this string will be null terminated" into the array `carr` starting at the address of `carr[20]`.

You can also copy a string at a certain address to another address. For example, you may want to copy the string beginning at the address of `carr[34]` to wherever the character pointer `cp` points:

```
> da cp
2482E5   t  h  i  s     s  t  r  i  n  g     i  s     t
2482F5   o  o  l  <  <  <  <  <  < \0 \0 \0 \0 \0 \0
248305  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
248315  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
> strcpy cp carr[34]
> da cp
2482E5   l  l     b  e     n  u  l  l     t  e  r  m  i
2482F5   n  a  t  e  d \0  <  <  <  <  < \0 \0 \0 \0 \0
248305  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
248315  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
```

The **strcpy** command can be used within the debugger either to initialize strings or character arrays or to copy strings from one area of memory to another.

Section 7

Controlling Program Execution

CodePRobe enables you to exercise a high degree of control over program execution. It allows you to single-step your program (at either the assembly or C source level), step over or into function calls, set breakpoints at source line or instruction addresses, or go to a specific location.

This chapter describes the following commands for controlling program execution:

Go	Start the execution of the program you are debugging.
Proceed	Single-step your program or step through it a given number of times, executing any function calls to their completion (unless they contain a breakpoint).
Restart	Reload your program and reinitialize your static data, allowing you to start over "from the top."
Trace	Single-step your program or step through it a given number of times, causing a break to occur when executing a function call.

Break	Set a breakpoint at the specified location.
Breakpoint List	List breakpoints set at various location.
Breakpoint Enable	Enable a breakpoint that has been disabled.
Breakpoint Disable	Disable a breakpoint at a specified location.
Return	Cause the function currently executing to return to its calling function.
Where	Display a backtrace of the currently executing function, its caller, its caller's caller, and so on.

This section introduces the concept of breakpoints. A breakpoint is basically an address used to control program execution. Each time the program attempts to execute at the breakpoint, program execution stops.

7.1 Go (g)

The **go** command starts the execution of the program you are debugging. This command uses the following syntax:

> **go** [*location* [*number*] [**when** (*condition*)]]

Note that the parentheses around *condition* in both **go** and the **break** commands are typed in explicitly. The **go** command has three different parameters: *location*, *number*, and *condition*.

The *location* parameter indicates where execution should terminate. The *number* parameter indicates a pass count. The *condition* clause indicates the circumstances under which execution is suspended at *location*. (It is identical to the **when** clause described later in this section for the **break** command.)

In its most general form, the **go** command can be read as:

> Go until *location* has been executed *number* of times and stop the next time *location* is reached if *condition* is true.

When used without parameters, the command starts the execution of your

program, which runs to normal completion if no breakpoints have been set or encountered.

7.1.1 Number

A pass count is indicated with the optional *number* parameter. Execution is not terminated at *location* until that *location* has been executed *number* of times.

7.1.2 Location

The *location* parameter can refer to an address or line number in a source file. It is the place where you intend to set a breakpoint. A *location* can take any of the following forms:

> *line*
> *function*
> *function* *line*
> *module* ! *function* *line*

The *line* is normally a number referring to the line number where you want execution to stop. It can also be:

$	current line
e[ntry]	first instruction of a function
r[eturn]	last instruction in a function prior to its return

The *line* is always relative to the beginning of the module. The *line* parameter can be used in conjuction with the *function* parameter to describe a *location*. For instance, in the form:

> *function* *line*

line refers to the line number of the module containing *function*.

If *function* or *module* ! *function* is specified without a *line* parameter, the entry point to the function is assumed. When the specified line is reached, a message is displayed describing the source file name, the function name, and

the line number. If mixed or assembly mode have been specified, the information is preceded by the breakpoint address. For example if C mode is currently chosen:

```
> go main 46
t02!main line 48
> set source mixed
> go main 48
at 00C80018 t02!main line 48
```

Here execution has stopped at line 48 of the file *t02.c* and within the function *main*.

If the *module* is not specified, a search will be made among your modules (beginning with the current module) until *function* is found. If *function* is not specified, the currently executing function is assumed. In the unlikely event that you have defined two functions in separate modules with the same name and belonging to the same static storage class, then it is necessary to specify the *module*.

Alternatively, when debugging in assembly mode, you may want to place breakpoints at specific addresses. To do this you specify the *location* as an explicit address constant.

7.1.3 Condition

The *condition* parameter is used to further control program execution. When this option of the **go** command is used, the debugger can plant a "non-sticky" breakpoint at a specified location. This breakpoint is "non-sticky" in the sense that when you resume execution after *location* is reached, the breakpoint is deleted. As a consequence, if you already have a normal breakpoint at a particular location and execute the **go** command to that location, the original breakpoint is also cleared.

A *condition* is of the form:

(*variable* | *number*) rel_op (*variable* | *number*)

where rel_op is one of the simple relational expressions permitted in C:

```
<
>
<=
>=
==
!=
```

Consider the following loop:

```
for (i = 0; i < 10; i++)
    b[i]  = 10 * i + 5;
```

You may wish to stop at a point within the loop, but only after a certain number of iterations of the loop have been executed. Assuming that the **for** statement is on line 62, this can be accomplished with the following **go** command:

```
> go 63 when (i == 7)
t02!main line 63
```

This has the effect of planting a "non-sticky" breakpoint at line 63 of the current module. That is, the breakpoint will be triggered only if the **when** condition is true, and otherwise execution will continue past the specified line. For example, if you give the command

```
> go 63 when (i == 15)
```

the breakpoint would not be triggered. Breakpoints will be discussed in greater detail later in this section.

If you make any mistakes in syntax when entering the condition, errors are not detected at the point that you issue the **go** command. However, the first time the specified location is reached, the debugger attempts to evaluate the condition. If it cannot do so, the breakpoint is triggered and an error message is displayed. The message indicates the position in the condition where syntax rules has been violated. Thus, for example,

```
> go 63 when (i !< 1)
Processing the when clause:
(i !< 1)
    ^------------syntax error
t02!main line 63
```

In stating the condition, "!<" violates rules governing permitted relation symbols.

7.1.4 Examples

The following are examples of ways to use the **go** command:

```
go main                /* go to the entry point of function main */
go return              /* go to the end of the current function */
go 0xC80200            /* go to the address 0xC80200 */
go swap 23 when (i==5) /* go to line 23 in function swap when i equals 5 */
go swap 23 5 when (k>10) /* go to pass 5 at line 23 in the function swap when k is < ten */
```

7.2 Restart (res)

The **restart** command causes the program you are debugging to execute as though it were being invoked for the first time. This command causes your program to be reloaded and executed up to the entry point of main, just as when the debugger was first invoked. However, if the debugger was originally invoked with the **-s** option, the "go main" command will not be executed. Because the program is reloaded, all static data is reinitialized. All breakpoints will be saved and watch breaks on static and external variables will be saved.

The **restart** command takes the following form:

res[tart] *list of arguments*

The *list of arguments* is a sequence of parameters delimited by white-space delimited, just as they would appear on the command line when invoking the program. Whatever the program takes as arguments can be used in this sequence as if they were invoked from the command line.

The **restart** command cannot be used as part of a command list or as one of a sequence of commands separated by semi-colons. When used, it must occur alone on the command line.

The debugger responds to the **restart** command by displaying the point at which the program begins:

```
> restart myfile
myfile!main line 11 - entry
```

7.3 A Sample Program

In order to illustrate the commands that allow control over program execution, it's best to use a sample program. The one below is a very simple (but inefficient) program that first initializes an array of integers from 0 to 9 and then sorts the array into reverse order. It consists of several functions in three modules: SMAIN.C, SORT.C, and SWAP.C.

The following source code is provided on disk 4 of this compiler package, in the directory called *:examples/debugger*. You will basically need to follow two steps:

1. Compile using the command `lc -d3 -L sort smain swap`.
2. Debug the program by entering `cpr sort`.

```
/* SMAIN.C */

int array [10];

main()
{
  int i;

  init(array); /* initialize array[] */
  /* Keep swapping elements until sorted */
  while (sort(array) != 0)
     ;
}
```

```
/* SORT.C */

 /* Initialize an array of integers */

void init(ip)
int ip[];
{
   int i;
   int *p;

   p = ip;
   for (i = 0; i < 10; i++)
     *p++ = i;
}

/* Pairwise sort elements of an array */

sort(ip)
int *ip;
{
   int i, s;
   int *p;

   p = ip;
   s = 0; /* no swaps performed yet */
   for (i = 0; i < 10 - 1; i++)
      if (p[i] < p[i+1] ) {
         swap(&p[i],&p[i+1]);
         s = 1; /* indicate a swap took place */
      }
   return(s);
}
```

```
/* SWAP.C */

void swap(x,y)
int *x, *y;
{
   int tmp;

   tmp = *x;
   *x = *y;
   *y = tmp;
}
```

This program will demonstrate commands used to control program execution.

7.4 Single-Stepping

There are two types of commands you can use to single-step your program: **proceed** and **trace**. Each of these commands has two variations: one to single-step at the assembler level and the other to single-step at the C source level.

The **proceed** and **trace** commands are similar in that they both allow you to single-step into your program or step through it a given number of times. The difference lies in their handling of function calls.

The **proceed** commands step "over" function calls; if a function call is encountered during the execution of a **proceed** command, the function is executed without a break occurring (unless one has been deliberately set in the function). This is in contrast to a **trace** command, which steps *into* function calls.

7.5 Trace Commands (t and ts)

The **trace** commands are used to single-step your program or to step through it a given *number* of times. Each step is either a single assembly instruction or a single line of C code. The **trace** commands are:

> **t**[race] *number*
> **ts** *number*

The optional *number* indicates the number of steps to be executed; if absent, a single step is executed.

The **t** command is for single-stepping assembler instructions. The **ts** command stands for "trace source", and is used for single-stepping C code. Either **trace** command will "step into" any function calls encountered during the step. That is, the called function is entered and execution is suspended at its first instruction.

The actions of these commands depend on the current source mode of the

debugger. In source mode, the two trace commands act identically, causing the execution of one line of C source code up to any imbedded function call. In assembly or mixed mode, **t** executes one machine instruction; in mixed mode, **ts** executes a single C source line.

Both **trace** commands step "into" function calls. In source mode, if the current line contains a function call, then the first line of that function is executed and subsequent **t** commands execute lines within that function. In assembly mode, if the current line contains a JSR (Jump to SubRoutine) or a BSR (Branch to SubRoutine) instruction, then that instruction is executed and subsequent **t** commands execute instructions in the called function.

Using the example given earlier and starting from *main*, trace for several steps in source mode:

```
smain!main line 4 - entry
> t
smain!main line 7
> t
sort!init line 5 - entry
> t
sort!init line 9
> t 3
sort!init line 10
sort!init line 11
sort!init in line 10
> t
sort!init line 11
> t
sort!init in line 10
> t
sort!init line 11
```

The first **trace** command carries us to the `init` call in *main*. The second then moves us into the *init* function. The third illustrates the use of a numeric argument to execute several steps.

To step by a single machine instruction, you can set the source to either **asm** or **mixed**. For instance, you can set your source mode to **asm** by using the pull-down Options menu or issuing the following command:

```
set source asm
```

Similarly, you can set your source to **mixed** with the following command:

```
set source mixed
```

At this point, you can enter the **t** command to step by an assembler instruction.

If source mode is set to **asm** and no line information is available, using the **ts** command is regarded as an error and an appropriate error message is displayed. In mixed mode, the **t** command causes stepping by machine instruction while the **ts** command causes stepping by C source line.

7.6 Proceed Commands (p and ps)

The **proceed** commands correspond closely to the **trace** commands. Like them, the **proceed** commands are used to single-step your program or to step it a given *number* of times. Each step is either a single assembly instruction or a single line of C code.

Unlike **trace**, however, a **proceed** command steps "over" function calls rather than stepping into them. That is, if a call to a function is encountered in the process of the step, the function is not stepped into as in the **trace** commands.

The **proceed** commands are:

> **p**[roceed] *number*
> **ps** *number*

Again, *number* indicates the number of steps to be executed. The **p** and **ps** commands are analogous to the **t** and **ts** commands with respect to source mode.

Using the previous example, notice how the results of the **proceed** commands differ compared to the **trace** commands (c source mode is used again):

```
> restart
smain!main line 4 - entry
> p
smain!main line 7
> p
smain!main line 9
> p
smain!main line 11
```

Here a single step has been made for each source line (7 and 9) of the function *main*. Function calls were made in the process of this stepping, but none of the called functions were entered.

The sort of single stepping you are most likely to use is of the **proceed** type. Therefore, **CPR** simplifies your use of the **p** command by allowing you to use the return key instead of entering this command. Pressing the return key is equivalent to entering the **p** command. Note how this works in our example:

```
> restart                    /* C Mode! */
smain!main line 4 - entry
>
smain!main line 7
>
smain!main line 9
> go sort
sort!sort line 18 - entry
>
sort!sort line 22
>
sort!sort line 23
> p 3
sort!sort line 24
sort!sort line 25
sort!sort line 26
> p 2
sort!sort line 27
sort!sort in line 28
```

Using the **ps** command in assembly mode is regarded as an error if no source line information is available, while **p** can be used in mixed or assembly mode

to single step by machine instruction. The **ps** command can be used in mixed mode to step by C source line.

7.7 Breakpoints and Actions

Aside from the commands used to display data such as **dump** and **display**, you will probably find the breakpoint commands of greatest use in debugging your programs. A breakpoint is an address at which program execution stops whenever encountered. The **break** command allows you to suspend the execution of your program at any point you choose. As an alternative to suspending execution, the **break** command allows you to execute a set of debugger commands at the breakpoint.

By setting conditional breakpoints or by using a pass count, you can suspend program execution at just the right stage of your program. This means, for example, that you can set a breakpoint at a line within a loop so that the program stops after going through 500 iterations of the loop. By triggering a breakpoint in this way, you are able to easily reach the point in the program which you wish to examine.

Besides breakpoint setting, additional commands allow you to manage your breakpoint list. These provide you with the ability to clear a breakpoint, temporarily disable a breakpoint then enable it again, or list the current breakpoints.

7.8 Breakpoint List (bl)

The **breakpoint list** command allows you to list all of your current breakpoints.The **breakpoint list** command is used repeatedly in the examples for other **break** commands. The format for the display of a breakpoint in the list is:

> *n* **b** *address location* (*number*) **when** (*condition*) {*action list*}

where *n* is the breakpoint number, *address* is the address of the breakpoint, *location* is the location in **module!function** line format, *number* is the pass count, *condition* is the **when** condition for the breakpoint, and *action list* is the list of actions to be executed at the breakpoint. For example:

```
> restart
smain!main line 4 - entry
> ex setbreaks
executing commands from setbreaks
> bl
1  b 00C8002E sort!sort entry
2  b 00C8004C sort!sort 25
3* b 00C8001B sort!init 11 when (i == 5)
4  b 00C80055 sort!sort 26 (5) (d p[:])
```

Here you can see that four breakpoints have been planted. The first is an unconditional breakpoint at the entry point to the function *sort*. The second is an unconditional breakpoint at line 25 of *sort*. The third is a conditional breakpoint set on line 11 of `init` that is triggered when the variable `i` has the value 5. The last breakpoint is a breakpoint with a pass count of 5. This breakpoint is triggered only after line 26 is executed five times.

The asterisk (*) beside the breakpoint number of the third breakpoint indicates that the breakpoint has been disabled. Notice that the line number is relative to the top of the source file even though a breakpoint at a given line is specified by referring to a function defined in that file. Thus, `init 11` refers to line 11 of the source file containing the function *init*.

Running the program in source mode at this point yields:

```
> go
sort!sort line 18 - entry
> go
sort!sort line 25
> go
sort!sort line 25
```

If we now clear the first, second, and fourth breakpoints and give the **go** command, the program should run to completion since the third breakpoint is disabled.

```
> bc 1 2 4
> bl
3* b 00C80024 sort!init 11 when (i == 5)
> go
```

```
program completed execution
```

Breakpoints that have been cleared are deleted from the breakpoint list. Each time you set a breakpoint, a new breakpoint number is created for that breakpoint. Such numbers are not reused, so if you define a number of breakpoints and clear several others, you can expect to see holes in the breakpoint list rather than consecutively numbered breakpoints.

7.9 Break (b)

The **break** command allows you to suspend the execution of your program at any point you choose or to execute a set of debugger commands instead of suspending execution. The **break** command has two forms:

b[reak]

b[reak] *location* [*number*] [**when** (*condition*)] [{*cmd_list*}]

The *cmd_list* parameter is a sequence of debugger commands enclosed in braces { } and separated by semicolons. In the first form, an unconditional breakpoint is set at the current location (which in source mode is the current line). This is handy when you have stepped to a certain point in your program and want to set a breakpoint quickly at the current line:

```
> go sort
sort!sort line 18 - entry
>
sort!sort line 22
>
sort!sort line 23
>
sort!sort line 24
> b
> bl
2  b 00C80044 sort!sort 24
> go
sort!sort line 24
```

Notice that in this case we have single stepped the program by using the return key (the same as a **proceed** command).

The second form of the **break** command allows you to set a breakpoint at a specified line number and to ensure that the breakpoint is not triggered until that line has been executed a specified number of times.

For example, suppose you want to set a breakpoint at the `if` statement in the `sort` routine but want the breakpoint triggered only after the fifth time that statement is executed. You can use the following:

```
> b sort 25 5
> bl
1  b 00C80055 sort!sort 25  (5)
> go
sort!sort line 25
> d i
4
```

The fact that the value of `i` is 4 indicates that the loop has already executed five times. You will find the pass count feature to be especially useful in debugging programs containing loops that iterate many times.

7.9.1 Conditional Breakpoints

Although breakpoints with pass counts are a kind of conditional breakpoint, the **break** command also allows for a more generally useful kind of conditional breakpoint. Using the **when** option, you can specify a condition that depends upon the values of variables in your program. You might, for example, want to stop at the `if` statement in *sort* when the value of `p[i]` is 8:

```
> b sort 25 when (p[i] == 8)
> go
sort!sort line 25
> d p[i]
8
```

Any *variable* or *number* can be used as an operand in the *condition*. Remember that a *variable* may be not only a simple identifier but also a complex expression involving reference to an array element, a structure or union member, or an enumeration constant.

Of course the pass count can be used in conjunction with the **when** clause. Both the pass count and the when clause conditions must be met. You may, for example, want to see if there is an instance where the array element is equal to the array index after three executions of a given line:

```
> b sort 25 3 when (p [i]  == i)
> go
sort!sort line 25
> di i
00C80B46       2
> di p [i]
00C8036C       2
```

7.9.2 Attaching Actions to Breakpoints

Instead of just stopping at a breakpoint, you may want to have the debugger perform one or more actions. You can do this by appending a list of actions to your **break** command. If you want to stop at line 25 of the *sort* function and have the debugger automatically display the value of `p [i]`, the value of `i`, and the address of `p`, you would use the following:

```
> b sort 25 {d "p[i]  = ", p[i] ; d "i = ",i; d "&p = ", &p}
> go
sort!sort line 25
p [i]  =  0
i =  0
&p =  0x00C80B4A
> go
sort!sort line 25
p [i]  =  0
i =  1
&p =  0x00C80B4A
> go
sort!sort line 25
p [i]  =  0
i =  2
&p =  0x00C80B4A
```

In fact, you can arrange for the debugger not to stop at the breakpoint but to continue:

```
> b sort 26 when (i < 3) {di p l 20; go}
> b sort 25 when (i >= 3)
> bl
1  b 00C80063 sort!sort 26 when (i < 3) {di p l 20; go}
2  b 00C80055 sort!sort 25 when (i >= 3)
> go
sort!sort line 26
00C80368:        0         1         2         3
00C80378:        4
sort!sort line 26
00C80368:        1         0         2         3
00C80378:        4
sort!sort line 26
00C80368:        1         2         0         3
00C80378:        4
sort!sort line 25
```

In this way, you can actually see the sort taking place and don't need to type in the **go** command repeatedly.

Your action list may be quite complex, and it may even contain commands to set breakpoints. For example:

```
> b sort 26 when (i > 5) {di p l 40; b sort 26 when (i == 8); bl; go}
> bl
1  b 00C80063 sort!sort 26 when (i > 5) {di p l 40; b sort 26 when (i == 8); bl; go}
> go
sort!sort line 26
00C80368:     1          2          3          4
00C80378:     5          6          0          7
00C80388:     8          9
2  b 00C80063 sort!sort 26 when (i == 8)
sort!sort line 26
> di p l 40
00C80368:     1          2          3          4
00C80378:     5          6          7          8
00C80388:     0          9
```

Here a breakpoint is set at line 26 of `sort` and is to be triggered when `i` has a value greater than 5. When triggered, the breakpoint action displays the first ten elements (40 bytes) of the array referenced by `p`, sets a new breakpoint at the same line, lists the new breakpoint, and continues execution. The new breakpoint is then triggered when its **when** condition is satisfied.

This nesting of breakpoints within action lists is a powerful feature of the debugger. Since nested breakpoints may become complex, you should take

great care in using them or you may become disoriented as to your position in the program when some breakpoints replace themselves with other breakpoints.

You must also be aware that a breakpoint with an action may have an effect (and perhaps an unexpected one) on program execution. To consider a simple case, suppose you set a breakpoint on line 26 of `sort` and attach to it an action that will alter the value of `i`:

```
> b sort 26 when (i == 5) {e i = 2}
> go
sort!sort line 26
> di i
00C80B46:        2
> go
sort!sort line 26
> di i
00C80B46:        2
```

In this simple case you have managed to create an infinite loop by means of your breakpoint and action list! Such a simple mistake is easy to find, but if you have planted a number of breakpoints or if your breakpoints are highly nested, such an error might be very difficult to detect. This example also demonstrates that the action for a breakpoint is executed before control is returned to you: by the time you examine the value of `i`, it has already been set to its new value by the **enter** command.

7.10 Break Clear, Disable, and Enable (bc, bd, and be)

Sometimes you will set a breakpoint and wish that you had not. You may want to remove it entirely or you may want to remove it just temporarily. The **breakpoint clear**, **breakpoint disable**, and **breakpoint enable** commands allow you to manage your breakpoints in these ways. The syntax of these commands is:

bc[lear] *bp_list*

be[nable] *bp_list*

bd[isable] *bp_list*

where *bp_list* is a list of breakpoints–a sequence of numbers denoting breakpoints and separated by blanks. This parameter may be in any of the following forms:

*

number
number number . . .
l[ast]

The first form denotes all current breakpoints, the second and third forms denote lists of breakpoints by breakpoint number, and the final form denotes the last breakpoint created.

Suppose that you have the following breakpoints:

```
> bl
1  b 00C80014 sort!init 10
2  b 00C80044 sort!sort 24 {d i}
3  b 00C80063 sort!sort 26 when (i > 5) {di p l 40}
4  b 00C80008 smain!main 7
```

Now you execute your program past several of these breakpoints:

```
> go
smain!main line 7
> go
sort!init line 10
> go
sort!sort line 24
-123
> go
sort!sort line 26
00C80368:        1           2           3           4
00C80378:        5           6           0           7
00C80388:        8           9
> go
sort!sort line 26
00C80368:        1           2           3           4
00C80378:        5           6           7           0
00C80388:        8           9
> go
```

```
sort!sort line 26
00C80368:        1            2            3            4
00C80378:        5            6            7            8
00C80388:        0            9
```

At this point you decide that the third breakpoint is not productive since it forces you to stop frequently in an inner loop. But you do not want to delete it since you may later need the information it displays. Also you notice that the first breakpoint will not be triggered again, and it did not do much in the first place, so you decide to remove it.

To remove a breakpoint from the breakpoint list, you use the **breakpoint clear** command, and to temporarily disable a breakpoint you use the **breakpoint disable** command:

```
> bc 1
> bd 3
> bl
2  b 00C80044 sort!sort 24 {d i}
3* b 00C80063 sort!sort 26 when (i > 5) {di p l 40}
4  b 00C80008 smain!main 7
```

A breakpoint that is cleared is removed from the list and cannot be reactivated without using the full **break** command and typing in the breakpoint again. However, a breakpoint that is disabled is marked as disabled (by means of an asterisk) and can be reactivated at a later time by means of the **breakpoint enable** command:

```
> be 3
> bl
2  b 00C80044 sort!sort 24 {d i}
3  b 00C80063 sort!sort 26 when (i > 5) {di p l 20}
4  b 00C80008 smain!main 7
> go
sort!sort line 24
9
>go
sort!sort line 26
00C80368:        2            3            4            5
00C80378:        6            7            1            8
00C80388:        9            0
```

Next you might choose to disable both 2 and 3 and run the program to completion:

```
> bd 2 3
> bl
2* b 00C80044 sort!sort 24 {d i}
3* b 00C80063 sort!sort 26 when (i > 5) {di p 1 20}
4  b 00C80008 smain!main 7
> go
program completed execution
```

7.11 Return (ret)

There may be circumstances in debugging a program where you want to return immediately from the current function without executing the remainder of the function. Such a case might arise if the current function is a lengthy one or if it will call a large number of functions before it returns. In addition, for testing purposes you might want to "stub out" a function that is a part of your program; a simple way to do this from within the debugger is to return from the function as soon as it is called.

The **return** command causes the function currently executing to return to its calling function. This allows you to determine when and what value a function returns. The syntax of the **return** command is:

> **ret**[urn] [*variable* | *address* | *number*]

The return value specified must be a scalar quantity (not a structure or union). Currently the **return** command does not support the return of aggregates. Examples of **return** commands include:

```
return 1
ret i
ret p[i]->txt
ret stringptr
return
ret #0x00C80200
```

When a **return** command is given, the current function is returned immedi-

ately to the calling function. The return value, if any, is returned to the calling function, and execution is suspended as though a breakpoint were triggered in the calling function after the call was made. For instance, take the code fragment:

```
i = 5;
j = func(i);
if (j < 2)
    j++;
```

Assume that the **return** command is issued during the execution of `func` in the form `ret 7`. Execution will stop in the `if` statement at the point where `func` returns. You can then single step to the next statement.

An attempt to return a value from a function declared "void" is treated as an error, and a suitable message is displayed. Similarly, an error message is displayed if you use a simple **return** command (without a parameter) from within a function declared as having a return value. In all other cases, the return value is cast (if necessary) to the correct type and returned to the calling function.

7.12 Where (whe)

You can use the **where** command to determine where you are in terms of function execution. It provides a stack backtrace, displaying the currently executing function first, followed by its caller, its caller's caller, and so on. Each line in the backtrace contains a function name, lists the values of parameters with which that function was called, and gives the source module and line number at which each call has been made.

For example, the **where** command may produce the following results:

```
> where
swap!swap (x=0x00C85400,y=0x00C85404) line 6
sort!sort (ip=0x00C85408) line 26
smain!main () line 4
```

The last (bottom) entry in the list of called functions is *main*.

Section 8

Using Watches and Watch Breaks

The **watch** commands are exceptionally powerful features of **CodePRobe**. The watch commands allow you to set, list, disable, enable, and clear watches and watch breaks. A *watch* (also called a *watch point*) allows you to monitor a variable, an address, or a range of memory in order to see when the value of the variable changes or the area of memory is altered. A *watch break* is similar to a watch but acts as a breakpoint if the variable or area of memory changes.

This section describes the following watch commands:

Watch List	List all watches and watch breaks that have been set and not cleared
Watch	Set a variable or area of memory to be watched
Watch Break	Set a breakpoint when a watched variable or area of memory changes
Watch Clear	Clear one or more watches and watch breaks

Watch Disable Disable watches or watch breaks

Watch Enable Enable watches or watch breaks previously disabled by the **watch disable** command

Like breakpoints, watches and watch breaks remain in memory until you clear them or exit **CPR**. You may restart the program being debugged and still retain these watch settings. This allows you to run through the program again and again without resetting the watches and watch breaks.

Watches and watch breaks are displayed in the watch window. This window is updated when the debugger returns control to the user (as at a breakpoint), at which point the new values of the watched areas are displayed. You can also display the values of the watched areas by using the **watch list** command. When a watch break is triggered, the message:

```
break (watch # n)
```

is displayed, where `n` refers to the numbered entry in the watch list. This message is followed by the usual breakpoint message describing the location at which the break has occurred. Watches provide a convenient way to watch memory on a continuing basis. It is often easier to examine the watch window instead of entering a **display** or **dump** command after every step or breakpoint.

Watch breaks can help you locate problem areas with precision. Often a variable or block of memory may be altered or corrupted at an unknown point in a program's execution. It can be very difficult to pinpoint the exact location where the corruption took place. By setting a watch break on the variable and executing the program, the problem can be isolated quickly.

Before describing the **watch** and **watch break** commands in detail, let's look at a command used to follow the results of setting watches and watch breaks: the **watch list** command.

8.1 Watch List (wl)

The **watch list** command displays a list of all watches and watch breaks that have been set and not cleared. The syntax for this command takes no parameters:

wl[ist]

Issuing a **watch list** command displays the following information:

- the watch number
- the ! character (if it is a watch break)
- the * character (if the watch or watch break has been disabled)
- the variable, address, or range at which the watch is set
- the value(s) at that address or range

For example, the **wl** command may produce the following results:

```
> wl
1   [sort] i : 0
2!  [sort] p [5]  : 3
3   tmp : 10
4 * 0x260370 : 61 6C 73 64 6B 6A 00 00 08 00 09 00 00 00 00 00 00 00 00 00
```

In this example, three watches and one watch break have been set, and the last watch has been disabled. For variables not defined in the module currently executing, the name of the variable is prefixed with the name of the function in which it is defined.

As your program executes, you can use the **watch list** command or the Watch Window to monitor the changes of the watched data.

8.2 Watch and Watch Break (w and wb)

The **watch** command (w) is used to set watches, while the **watch break** command (wb) is used to set watch breaks. The syntax of these commands is:

w[atch] (*variable* | *address* | *range*)
wb[break] (*variable* | *address* | *range*)

where *variable* can be a simple variable, a structure, a union, a member of a structure or union, an array, or an array member; *address* can be a constant, a C pointer variable, a register, or the result of prefixing "&" to a C identifier of the proper type; and *range* is any contiguous area of memory.

With the **watch** command, the new value of the watched object is displayed in the watch window whenever control is returned to you (for example, at a breakpoint). If the Watch Window is not open, you can use the **watch list** command as a way of determining the new value.

Watches (and watch breaks) can be set on structures or arrays by name. When this is done, the specification of the aggregate name is treated as an implicit range and a watch is set on that range. Assume the following declarations are in effect:

```
struct X
 {
           int i;
           char a[20] ;
           double d;
 } x;

char string[80] ;
```

A watch can be set on ranges as:

```
> w x
> w string[5] L 10
> w string[17] string[19]
```

Watches and watch breaks can be set on formal or automatic variables. When this is done, the watch is set on the *variable* rather than just an address. In this case the watch is in effect only as long as the function defining variable is executing. If you set a watch on a formal or automatic variable, the watch list displays the watch as "not active" when execution is taking place outside its defining function. When the function is re-entered, the watch is re-activated.

It is possible to specify in a purely symbolic way a rather bizarre range. For

example, suppose that `aut` is an automatic variable and `sta` is a static variable. Issuing the command:

```
> w aut sta
```

sets a watch on the memory between the address of `aut` and the address of `sta`. Of course, the address of `aut` changes as its function is entered, executed, and exited (possibly a number of times). In the case of such "artificial ranges," the debugger translates the symbolic names into their current absolute address equivalents and sets the watch on the absolute address range. It is this absolute address range that is displayed in the watch list. In general, this sort of translation from symbolic range to absolute range takes place when the two variables you use to specify the range do not resolve to addresses within the same aggregate (structure, union, or array).

While watch breaks can be useful in monitoring program execution, they are expensive in terms of execution time. You should expect your program to execute much more slowly when watch breaks are enabled. (This is not true of watches since the information for these is updated only when control is returned to you by the debugger.) Watch breaks should be disabled whenever possible to speed execution. One way you can do this is to enable and disable a watch break dynamically depending on program conditions. A command sequence such as:

```
> wb i
> wd 1
> b sort 9 {we 1; go sort 20; wd 1; go}
```

installs a watch break (whose number in the example is assumed to be 1) and then sets a breakpoint that automatically enables the watch break, executes up to a certain point, disables the watch break, and continues execution. In this way, you can force the watch break to be active only when a certain portion of your program is executing.

8.3 Watch Clear, Disable, and Enable (wc, wd, and we)

The **watch clear** (**wc**), **watch disable** (**wd**), and **watch enable** (**we**) commands are used to clear, disable, and enable watches and watch breaks. These commands are similar to the corresponding commands for breakpoints. The syntax for these three commands is:

> **wc**[lear] *wp_list*
>
> **wd**[isable] *wp_list*
>
> **we**[nable] *wp_list*

where *wp_list* is a list of watch points–a sequence of numbers denoting watches and separated by blanks. Watch breaks may take any of the following forms:

> *
>
> *number*
> *number number* . . .
> l[ast]

The first form denotes all current watches, the second and third forms denote lists of watches by watch number, and the final form denotes the last watch created.

Note that disabling a watch really has no effect since the updated value of its watched area is displayed whether it is disabled or not. However, disabling a watch break has the same effect as disabling a breakpoint. That is, if a watch break has been disabled and the watched area is changed, then no break is triggered. Clearing watches and watch breaks is done in a manner similar to clearing breakpoints.

As a simple example, consider the following definitions and modifications of watches:

```
> wl
1   [sort] p : 0x00260368
2!* [sort] s : 0
3!   tmp : 6581
> wd 3
> we 2
> wl
```

```
1   [sort] p : 0x00260368
2!   [sort] s : 0
3!* tmp : 6581
```

In this example, one watch break (3) is disabled while another (2) is enabled. Notice that listing reflects this change by the shift in the asterisk.

Section 9

Other CodePRobe Commands

In addition to the commands previously discussed, **CodePRobe** offers a number of commands which may be useful in special instances. These commands include:

List	List source lines in one or more modules.
Unassemble	Display a portion of memory as M68000 assembler instructions.
Execute	Execute scripts within the debugger, usually to change your environment or to invoke a sequence of commands.
Hunks	Display the addresses and sizes of the application's segments as scatter loaded into memory by the AmigaDOS loader.

9.1 List (l)

The list command is primarily useful when in line mode, but it can be used in window mode to change the current source file or line range. In window mode, the more complicated forms of line ranges are meaningless. In this case, you simply specify a single line and **CPR** displays your source centered in the Source Window about that line.

The **list** command can be used to list source lines in one or more modules. Its syntax is:

l[ist] *list_range*

The *list_range* is an expression of the form:

[*module*] *line_range*

If the optional *module* parameter is specified, only the first four of the following forms of *line_range* are permitted. Without the *module* parameter, any of these forms can be used.

Form	Meaning
*	current line
number	just line *number*
number1 L *number2*	lines *number1* through *number2*
number1 L +*number2*	line *number1* and next *number2* lines
+*number*	current line and next *number* lines
+*number1* L +*number2*	lines from (current + *number1*) through (current + *number2*)
-*number*	previous *number* lines and current line
-*number1* L -*number2*	lines from (current - *number1*) through (current - *number2*)
-*number1* L +*number2*	lines from (current - *number1*) through (current - *number2*)

Unless otherwise specified, the current module is assumed. You can list a single line or a range of lines:

```
> l 7
   7:      init(array);   /* initialize array [ ]  */
> l 3 L 9
   3: main()
   4$
   5:    int i;
   6:
   7:      init(array);   /* initialize array [ ]  */
   8:      /* Keep swapping elements until sorted */
   9:      while (sort(array) != 0)
```

Notice that the current line is marked with a "$."

The range of lines you list can be specified in a relative way: you can list the next n lines, the previous n lines, or a neighborhood of lines around the current line:

```
> l +4
   4$
   5:    int i;
   6:
   7:      init(array);   /* initialize array [ ]  */
   8:      /* Keep swapping elements until sorted */
> l -2
   2:
   3: main()
   4$
> l -2 L +5
   2:
   3: main()
   4$
   5:    int i;
   6:
   7:      init(array);   /* initialize array [ ]  */
   8:      /* Keep swapping elements until sorted */
   9:      while (sort(array) != 0
```

If you want to look at the source in another module, it is necessary only to specify that module prior to specifying the range of lines to be listed Since the current line is in the current module (that is, that module which currently

is being executed), no current line is displayed when listing another module. Likewise, those types of *list_range* that are only meaningful relative to the current line cannot be used when listing sources from a module other than the current one.

9.2 Unassemble (u)

The **unassemble** command displays a portion of memory as M68000 assembler instructions. The syntax is as follows:

> **u**[nassemble] [*location* [*location*]]

A location can be either a line number, a function name, or an absolute constant address. If two locations are provided, they are treated as a range.

By default, the command begins disassembling at the current program counter. If subsequent **unassemble** commands are performed before the application is given control by a **trace**, **proceed**, or **go** command, it will continue disassembling from where it finished the last invocation.

If source line number information is available for the range of memory being disassembled, **unassemble** will display the C source line before displaying the corresponding instructions. By default, it will display the number of assembler instructions generated for one C source line being displayed.

If no source information is available for the range of memory being displayed, **unassemble** will display the default number of instructions as specified in the Option menu. By default this number is 4.

The output of the **unassemble** command is equivalent in format to assembler mode output displayed in the Source Window. It will consist of three or four columns depending on the setting of the "Instruction Bytes" option which is controlled by the **set ibytes** command (see Section 3). The following is an example of unassemble output with **instruction bytes** on and source available:

```
> u

0025F950 48E70130          MOVEM.L    D7/A2-A3,-(A7)
0025F954 BFEC0004          CMPA.L     0004(A4),A7
0025F958 65001BD6          BCS        00261530
```

The first field contains the hexadecimal address of the instruction being disassembled. The second field is a hexadecimal dump of the actual bytes comprising the instruction. The third field consists of the M68000 mneumonic for the given opcode. The fourth field consists of the operands to the instruction. If the instruction bytes option is turned off, the second field is not displayed.

9.3 Execute (ex)

In addition to profile scripts, you may execute scripts at anytime within the debugger. This may be useful if you wish to change your environment or if you have a common sequence of commands that you frequently execute.

The **execute** command is used for this purpose. Its syntax is:

ex[ecute] *file*

where *file* is a file name, possibly including drive, path, and extension as needed. The **execute** command reads the named file of **CPR** commands and executes those commands. If *file* cannot be found, **CodePRobe** appends a ".cpr" extension to it and tries again. Thus, you can place a set of **CPR** commands in a file with a .cpr extension and simply use the root file name (minus the extension) in the **execute** command.

If the file *cmds.cpr* consists of the commands

```
b sort
bl
t
t
```

then the **execute** command applied to this file would yield

```
> ex cmds.cpr
6  b 00C80037 sort!sort entry
smain!main line 7
sort!init line 5 - entry
```

The debugger commands themselves are not echoed as part of the display unless echo mode is active.

If you are debugging a program frequently and want to set a number of breakpoints at the same places, you can place all of your breakpoint commands in a file and then use the **execute** command on this file. In this way you will avoid excessive typing of the same commands each time you want to use the debugger.

A restriction on the use of this command is that if it is used in a command list, it must occur as the last command in the list.

9.4 Hunks (hu)

The **hunks** command displays the addresses and sizes of the application's segments as scatter loaded into memory by the AmigaDOS loader. The **hunks** command takes no options and can be abbreviated to two characters. Example:

```
>hu
Hunk  Address     Size
 0    002636D8    0x2DF8 (11768)
 1    00228268    0x470 (1136)
```

The size is displayed first in hexadecimal format, followed by its decimal format in parentheses.

Section 10

Multi-Tasking Applications

CodePRobe can debug applications that spawn additional tasks. These applications are designed to take advantage of the Amiga's multi-tasking capabilities. This section will look at the commands used to debug multi-tasking applications including:

Tasks	Display all tasks under the debugger's control.
Set Task	Examine or modify the state of some task besides the one currently at a breakpoint.
Deactivate	Prevent a task from running, even when a "go" is performed from another "current" task.
Activate	Reactivate a task affected by the **deactivate** command.
Detach	Free a task from debugger control.
Catch	Place a task under debugger control.
Symload	Change the load module used for collecting debug information while the debugger is active.

The discussion concludes by focusing on how to design applications with debugger compatibility in mind.

CPR is able to debug multi-tasking applications by intercepting every call to AddTask( ) and determining the identity of the calling task. If the calling task is under debugger control, the new task will also be brought under debugger control. Any task under **CPR**'s control can be stopped by a set breakpoint or by single stepping. Also, if you type Ctrl C while an application is running, all active tasks under debugger control will be stopped.

10.1 How Tasks are Handled

It is important to understand the way tasks are handled when a breakpoint is hit or when a task is being single stepped. If several tasks are running under the debugger when a breakpoint is reached, all other tasks on the debugger's list are stopped as well the one that hit the breakpoint. This guarantees that the state of the application remains consistent while the user examines the task. The debugger will return to the user with the state of whichever task actually hit the breakpoint. That is, the "current module," "current line," and the registers displayed will all be for the breakpointed task. The name and address of the Task Control Block for the current task are always displayed in the Dialog Window's title bar.

10.2 Tasks (ta)

A number of commands are provided to aid in manipulating tasks. The **tasks** command displays all tasks under the debugger's control including the debugger itself:

```
>tasks

Address   Type  Pri  State    SigWait   StackPtr  Debug  Name
00C513B8   13    0  Waiting  80000000  00C551CC   act   multi
00C1C328   13    0  Running  00000000  00C1C200   act   cpr
```

Note that there are eight fields (columns) displayed by this command:

Address Contains the address of the Task Control Block.

Type Indicates the type of node that begins the task control block. The number 13 is used for processes, while a 1 is used for simple tasks.

Pri Contains the priority of the task.

State Indicates the process state.

SigWait Displays the SigWait field of the Task Control Block. This field indicates which signal bits the task may be waiting on.

StackPtr Indicates the current position of the stack pointer. Note that this will be somewhat different from the value displayed in the sp register. The value displayed here includes any information placed on the stack by the trap handler or exception handler associated with this task.

Debug Indicates the task's status with the debugger. The values are **act** for active tasks **inact** for inactive tasks. Tasks not under the debugger will have no information in this field.

Name Refers to the name of the task as specified in the task's node structure.

For more information on tasks and any of the fields (except the debug field) just discussed, consult the *Amiga ROM Kernel Manual: Exec*.

The **tasks** command will accept an option called **all**. This option causes all tasks in the system to be displayed. This can be useful if you wish to "catch" a task that isn't currently under debugger control. (For more details, see the discussion on the **catch** command later in this section.)

10.3 Set Task (se t)

If several tasks are currently running under the debugger, you may wish to examine or modify the state of some task besides the one currently at a breakpoint. The **set task** command is provided for this purpose.

When a new task is made "current" by this command, you will observe a new set of registers. Highlighting of changed registers in the Register Window may be misleading since the debugger only keeps track of changes while

stepping through a single task. The module displayed in the Source Window may change, or more likely, if the task was in a resident or linked library, assembler lines will be displayed. It may be possible to get information as to the calling sequence of the task by using the **where** command, however since assembler routines (including the Amiga's Resident Libraries) don't use the A5 register as a frame pointer, **where** can only make an intelligent guess.

If you wish, you may modify any registers, condition control register (CCR) flags, or stack variables, just as in the breakpointed task. You may even single step through the code. However, **a word of caution is advised!** A breakpoint is implemented by placing an illegal instruction at the desired location. All tasks under the debugger's control have a trap handler attached to catch the illegal instruction and report back to the debugger. A breakpoint in any shared code will cause an illegal instruction in any task that executes that code. If you must place a breakpoint in shared code, such as a resident library under test, be sure that any task that might open it is under debugger control. The standard system libraries such as Exec and Intuition must *never* have breakpoints installed in their code!

The **set task** command, like all commands that manipulate tasks, takes a *task-ID* as an argument:

> **se**[t] **t**[ask] *task-ID*

A *task-ID* can be either the name of the task or the address of the task control block. If the name is used, it should be unique (otherwise the *first* task in the debugger's task list will always be selected). If the name contains blank, "white" space, the entire name must be surrounded by double-quotes.

Task addresses are generally entered in hexadecimal with a `0x` prefix. Task names and addresses can be determined with the **tasks** command. Some sample commands are:

```
>set task Child
>set task 0xC85428
>set task "Child of multi"
```

NOTE: task names are case-sensitive.

10.4 Activate and Deactivate (a and d)

While debugging a multiple task application, you may wish to temporarily stop one or more tasks completely. The **deactivate** command allows you to prevent a task from running, even when the **go** command is performed from another "current" task. Later, the task can be reactivated by using the **activate** command. Both commands take a *task-ID* as an argument.

> **a**[ctivate] *task-ID*
> **d**[eactivate] *task-ID*

10.5 Detach (det)

At times, you may wish to allow one or more tasks spawned by a task under the debugger to run freely. For example, a task designed to respond to Intuition events may cause the system to "lock up" if it doesn't respond to menu events quickly. It may not be desirable to allow such a task to be stopped when other tasks hit a breakpoint. Such a task can easily be freed from debugger control by using the **detach** command.

To use the **detach** command, first set a breakpoint at the entry point to the task. When the breakpoint is reached, use the **tasks** command to identify the address or name of the task (you may know this based on your code). Then enter the **detach** command. The syntax is the same as the other task manipulation commands previously discussed.

> **det**[ach] *task-ID*

A few cautionary words are in order regarding the use of the **detach** command. Once the task is free from the debugger, it can no longer handle breakpoints. Therefore, be sure that all breakpoints in code reachable from a detached task have been removed. Second, the task will not be removed automatically if you quit the debugger. If the segment of code is unloaded or freed, the task may later attempt to enter it, providing you with a friendly visit from the guru.

If you later wish to re-attach a task to the debugger, this can be achieved using the **catch** command, the subject of the following discussion.

10.6 Catch (ca)

At times, you may find that a process or task not started under the debugger is behaving in an unpredictable or undesired manner. For instance, the task may be caught in an infinite loop. Another possibility is that the task is waiting on some message port for a message that will never come. The **catch** command is used to capture such "unruly" tasks. Its syntax is similar to those for other task commands:

ca[tch] *task-ID*

First, invoke **CodePRobe** with the **-n** option and the name of the load module with debugging information present. This option will load the debugger and the symbolic information, but it will not load the program into memory, or start up a process. After all, your intention is to capture an existing process or task, not to generate another one.

Second, when the debugger is up, use the command **tasks all** to find the name and address of the process in question.

The **catch** command will capture the offending task, and place it under debugger control. If the task is a process, this command will locate the code segments associated with the debug load module as available from the process structure. If the task is not a process, you will have to determine the address of the segment list and use the **set env** command to tell the debugger how to map the debug information with the actual load module. The **set env** command takes as an argument the address of a segment list pointer. For example:

```
set env 0xC08418
```

For more information on segment lists, see *The AmigaDOS Manual*.

10.7 Symload (sym)

Another command particularly useful for handling multi-tasking applications is the **symload** command. This command allows you to change the load module used for collecting debug information *while the debugger is active.*

This is useful when debugging several interacting applications that are invoked from separate load modules. Note that this command does not load any code into memory or spawn any new processes or tasks. The **symload** command takes as its argument the name of an executable object file containing debug information.

sym[load] *file*

10.8 Design Considerations for Debugger Compatibility

CodePRobe makes every effort to allow an application to run as it would while running outside of debugger control. However, in order to establish breakpoints and catch new tasks generated in a multi-tasking application, **CPR** must manipulate certain resources in the application's task structure. In particular, **CPR** redefines a tasks exception handler, exception data, trap handler, and trap data fields. If it is necessary for an application to redefine any of these fields for its own purposes, it must do so in the way prescribed by the *Amiga ROM Kernel Reference Manual: Exec*.

For exception handlers, this means that it must save the address of the original exception handler and exception data fields, and pass any unallocated exceptions back to those routines. Before invoking the original exception handler, be sure to restore the original exception data pointer in the task structure.

The same practice applies to trap handlers. In particular, all exceptions besides allocated traps should be passed to the original trap handler. In order to control multi-tasking, **CPR** performs a SetFunction on the AddTask function. This function should not be redefined by an application.

Appendix A

Error Messages

CodePRobe has a number of error messages that you might encounter for a variety of reasons. You might mistype a command or use incorrect command syntax. If this is the case, **CodePRobe** will inform you of the error and display a pointer to the section of the command where the error occurs. Other error messages pertain to the information or lack of information that **CodePRobe** has while processing commands. For example, if you have failed to compile some of the modules in your program with the correct debugging options or if you have failed to link the program with the debugging option of the linker, then **CodePRobe** will not have the necessary information to execute certain commands you may give it. The error messages used by **CodePRobe** are listed below in alphabetical order with brief explanations.

. . . is not a pointer

You have attempted to use the named expression in a context where a pointer (or address) is required, but it does not refer to one.

. or -> is invalid . . .

You have attempted to use the dot or arrow operators in a context where they are meaningless. These can be used only for structures or unions. The

expression preceding the -> must be a pointer to an object of the proper type (structure or union).

Accessing extern with unknown attributes - assuming int

You have made reference to an object that is not declared in the current modules, so **CodePRobe** does not know what type of object this is. It is assumed to be of type int unless you cast it (for example, in the **display** command) as an object of another type.

Auto variables do not exist on entry

This message is likely to be displayed if you have asked the debugger to display the value of an automatic variable before the function entry code for its function has been executed. Since the function entry code defines a stack frame in which automatic variables are stored, they cannot be accessed before this stack frame is defined.

Can't cast this object

This message may be generated when you attempt to use the **display** command with the *typename* or *tag* option. Certain objects cannot be used in a cast expression or, equivalently, as a *typename* or *tag* in the **display** command. These include bit fields, enumeration constants, and register names.

Can't dereference non-pointer

You have attempted to use a non-pointer as a pointer by dereferencing it with either a '*' operator (*p), an array index operation (p[i]), or an arrow operator (p->x). None of these operations can be applied to a non-pointer.

Can't find member . . .

The named member of a structure or union cannot be found. Most likely you are looking for it in the wrong structure or union.

Can't find required type

This message can occur if the *typename* or *tag* specified in the **display** command cannot be found.

Can't find variable . . .

The named variable cannot be found. Perhaps you have mistyped its name.

Can't take the address of . . .

You have attempted to take the address of something whose address cannot be taken in C. Remember that you cannot take the address of register variables, enumeration constants, or bit fields.

Can't take the address of a bit field

You have attempted to take the address of a bit field. This is prohibited in C.

End of source on line . . .

You have asked **CodePRobe** to access a source line that does not exist. The module does not have that many source lines in it.

Error opening source file . . .

An error occurred in opening the named source. This message is generated only under unusual circumstances: when the named source file has been opened previously but a more recent attempt to open it has failed. Perhaps you have removed the file between opens, or perhaps the program you are debugging has removed or renamed the file.

Function not in any module

You have referred to a function that is not defined in your program.

Function not in module specified

You have referred to a function as being in a certain module (by using the module!function notation) and there is no function of that name in the module.

Identifier truncated

Any identifier longer than the maximum identifier length (31 characters) is truncated to that length.

Input discarded: . . .

The input displayed by the message has been ignored. This can happen if you have given multiple debugger commands on one command line (separated by semicolons). The debugger has been able to execute one or more of these commands but is unable to execute a command found later in the list. The remainder of the command list is displayed.

Invalid command

You have given a command that **CodePRobe** does not recognize.

Invalid operation on function pointer

You have made an attempt to dereference a function pointer.

Invalid source for ENTER
Invalid target for ENTER

These messages can be generated because the source (target) parameter for the **enter** command refers to a global data item not declared in the current module. Since it is undeclared, the debugger does not know its size or other characteristics. Alternatively, the message can occur because the source or target parameter does not resolve to an address (or you have not used the form of the command that allows you to cast it as an address).

Invalid text after valid command

You have given a valid command, but there is superfluous text on the same line as the command.

Invalid type for array subscript

The type of the expression used as an array subscript must be one of the integral types. It cannot be a pointer, structure, union, or floating point expression.

Invalid type for watch

Watches cannot be set on constants, explicit string constants, or enumeration constants.

Line not in function specified

The line to which you have referred is not in the function you have named. Perhaps that particular line occurs within a different function in the same file.

Missing/invalid file name

Either you have failed to provide a file name where one is needed or the file name you have specified in not a valid file name.

Module cannot be found

You have referred to a module that is not part of your program. Perhaps you have mistyped the name of the module.

Module has a bad symbol table

The symbol table for the module has become corrupted. Try recompiling the module and relinking your program. Do not forget to use compiler debugging options and the linker debugging option.

Module has no line number/offset table

You have failed to compile the module with one of the compiler debugging options, so no information concerning source line numbers has been put in the object file or passed on to the linker. Recompile the module with at least the -d option and do not forget to use the /debug option of the linker when linking your program.

No code at this line

You have tried to go to, single step to, or set a breakpoint at a line where there is no code.

Restart failed

Your attempt at restarting the program has failed. Try quitting **CodePRobe** and invoking it again.

Should be an address

Some commands require certain of their parameters to be addresses. An address is rather broadly defined. See Section 5 for a detailed description of the *address* parameter.

Should be a scalar variable

Some commands require scalar variables (not structure or union variables) as parameters. You have used an aggregate variable in such a command.

Source file is empty

The source file has no code in it. Perhaps you have accidentally overwritten the file.

Source for module is unavailable

The source code for the module cannot be found. Remember that source files must be present in your current directory when you are running the debugger if you want to refer to their source lines. This message also can occur if there is no line number/offset table for the module, as when no compiler debugger option has been used to compile the module.

Symbol table not available

CodePRobe cannot find the symbol table for your program. Be sure that you have compiled your program with one of the compiler debugging options and linked it with the debug option of the linker.

ts/ps is invalid in assembler mode

You are not in C source mode but have told **CodePRobe** to single step by a C source line. This is not permitted. Set your source mode to C mode or mixed mode if you want to use either the **ts** or **ps** command.

Index

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Library Reference

Library Reference

Library Reference

SAS/C® Library Reference Manual

Version 5.10

Part of the SAS/C® Compiler for AmigaDOS™

SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513-2414
USA

SAS/C® Library Reference Manual

This document was produced using *HighStyle*®, the Lattice Document Composition System.

Lattice® Amiga C Library Reference Manual

Version 5.0

Part of the Lattice C Compiler for AmigaDOS

Lattice, Incorporated
2500 S. Highland Avenue
Lombard, IL 60148
USA

Subsidiary of SAS Institute Inc.

Lattice Amiga C Library Reference Manual

This document was produced using ***HighStyle***™, the Lattice Document Composition System.

Lattice Amiga C Library

This is a user's guide for the library functions and data names making up the Lattice AmigaDOS C Compiler package. The libraries form a collection of *external names* that can be referenced from within your C programs. Function names, of course, are intended to be called, while data names are intended to be read or written. You can also form a pointer to any external name via the ampersand (&) operator.

While the bulk of the library section describes library functions, the following data names are also included:

DOSBase	_timedata
errno	_MathTranBase
DiskfontBase	_fmode
GfxBase	_FPERR
IntuitionBase	_MNEED
MathBase	_MSTEP
msflag	_OSERR
_bufsize	_SLASH

Before detailing the library functions and data names, we include a combined index for your convenience and information. This index acts as a quick, yet comprehensive reference point to all of the library functions and data names described in this portion of the manual. If you are looking for a function to perform a particular task, it is often useful to scan through the one-line descriptions in the combined index. Page numbers are referenced with each entry.

The combined library index also references the portability classification of each function or data name. Most C programmers are somewhat concerned about portability, in the sense that they do not want to write software that locks them into a particular computer. They like to design general-purpose modules which can be re-used from one application to another. At the same time, C programmers are interested in writing programs that approach the performance of assembly language.

The Lattice library attempts to strike a reasonable balance between portability and performance. It includes the highly portable functions defined by the ANSI C Standards Committee along with a large number of functions that utilize AmigaDOS more efficiently or provide access to features that are peculiar to AmigaDOS.

To assist you in this decision, each data or function name has been given a portability classification as follows:

AmigaDOS The name can be used only under Commodore's AmigaDOS.

ANSI The name is defined in the proposed ANSI Standard for the C language, and our implementation is compatible with that definition.

LATTICE The name is in Lattice's portable library. That is, we intend to make it available on any system for which we provide a C compiler package, unless the system prevents us from doing so.

OLD The name was defined in earlier versions of the Lattice C Compiler and can still be used, although the newer ANSI and AmigaDOS functions are now preferred.

UNIX The name is defined in AT&T's UNIX System V, and our implementation is compatible with that definition.

XENIX The name is defined Microsoft's XENIX, and our implementation is compatible with that definition.

It's up to you to decide whether to write your programs using only the portable ANSI functions or to employ less portable but faster and easier facilities.

Library

NAME	DESCRIPTION	CLASS	PAGE
abort	Abort the current process	ANSI	L1
abs	Absolute value	ANSI	L2
access	Check file accessibility	UNIX	L4
acos	Arccosine function	ANSI	L251
argopt	Get options from argument list	LATTICE	L6
asctime	Generate ASCII time string	ANSI	L9
asin	Arcsine function	ANSI	L251
assert	Assert program validity	ANSI	L11
astcsma	AmigaDOS string pattern match (anchored)	AmigaDOS	L13
atan	Arctangent function	ANSI	L251
atan2	Arctangent of x/y	ANSI	L251
atexit	Set an exit trap	AmigaDOS	L14
atof	Convert ASCII to float	ANSI	L15
atoi	Convert ASCII to integer	ANSI	L17
atol	Convert ASCII to long integer	ANSI	L17
bldmem	Build a memory pool of specified size	OLD	L18
calloc	Allocate and clear Level 3 memory	ANSI	L19
ceil	Get ceiling of a real number	ANSI	L23
chdir	Change current directory	UNIX	L24
chkabort	Check for Break character	AmigaDOS	L25
chkml	Check for largest memory block	LATTICE	L26
chkufb	Check Level 1 file handle	LATTICE	L27
chmod	Change file protection mode	UNIX	L28
clearerr	Clear Level 2 I/O error flag	ANSI	L30
close	Close a Level 1 file	UNIX	L31
clrerr	Clear Level 2 I/O error flag	UNIX	L30
cos	Cosine function	ANSI	L251
cosh	Hyperbolic cosine function	ANSI	L251
cot	Cotangent function	ANSI	L251
creat	Create a Level 1 file	UNIX	L32
ctime	Convert time value to string	ANSI	L34
CXFERR	Low-level float error	LATTICE	L36
datecmp	Compare two AmigaDOS dates	AmigaDOS	L37
daylight	Daylight savings time flag	UNIX	L269
dfind	Find first directory entry	AmigaDOS	L40

NAME	DESCRIPTION	CLASS	PAGE
DiskfontBase	Disk font library vector	AmigaDOS	L71
dnext	Find next directory entry	AmigaDOS	L40
DOSBase	AmigaDOS Library Vector	AmigaDOS	L43
dqsort	Sort an array of doubles	LATTICE	L166
drand48	Random double (internal seed)	UNIX	L44
ecvt	Convert float to string	UNIX	L49
_assert	Failure exit for assert	ANSI	L11
_bufsize	Level 2 I/O buffer size	LATTICE	L259
_dclose	Close an AmigaDOS file	AmigaDOS	L38
_dcreat	Create or truncate AmigaDOS file	AmigaDOS	L39
_dcreatx	Create new AmigaDOS file	AmigaDOS	L39
_dopen	Open an AmigaDOS file	AmigaDOS	L42
_dread	Read from an AmigaDOS file	AmigaDOS	L47
_dseek	Re-position an AmigaDOS file	AmigaDOS	L48
_dwrite	Write to an AmigaDOS file	AmigaDOS	L47
_exit	Terminate with no clean-up	ANSI	L55
_fmode	Default level 2 I/O mode	LATTICE	L260
_FPERR	Floating Point Error Code	LATTICE	L261
_main	Standard preprocessing for the main module	LATTICE	L264
_MNEED	Minimum Dynamic Memory Needed	LATTICE	L263
_MSTEP	Memory Pool Increment Size	LATTICE	L265
_OSERR	DOS Error Information	AmigaDOS	L266
_SLASH	Directory separator character	LATTICE	L268
_tinymain	Special version of _main	LATTICE	L264
emit	Emit 68000 instruction word	LATTICE	L51
erand48	Random double (external seed)	UNIX	L44
errno	UNIX error number	UNIX	L52
except	Call math error handler	LATTICE	L146
exit	Terminate with clean-up	ANSI	L55
exp	Exponential function	ANSI	L57
fabs	Float/double absolute value	ANSI	L2
fclose	Close a Level 2 file	ANSI	L58
fcloseall	Close all Level 2 files	XENIX	L58
fcvt	Convert float to string	UNIX	L49
fdopen	Attach Level 1 file to Level 2	UNIX	L60

NAME	DESCRIPTION	CLASS	PAGE
feof	Check for Level 2 end-of-file	ANSI	L61
ferror	Check for Level 2 error	ANSI	L61
fflush	Flush a Level 2 output buffer	ANSI	L62
fgetc	Get a character from a file	ANSI	L63
fgetchar	Get a character from stdin	XENIX	L63
fgets	Get string from Level 2 file	ANSI	L64
fileno	Get file number for a Level 2 file	UNIX	L66
findpath	Locate file in the current path	AmigaDOS	L67
floor	Get floor of a real number	ANSI	L23
flushall	Flush all Level 2 output buffers	XENIX	L62
fmod	Compute floating point modulus	ANSI	L68
fmode	Change mode of Level 2 file	LATTICE	L70
fopen	Open a Level 2 file	ANSI	L72
forkl	Fork with arg list	LATTICE	L76
forkv	Fork with arg vector	LATTICE	L76
fprintf	Formatted print to a file	ANSI	L83
fputc	Put a character to a level 2 file	ANSI	L90
fputchar	Put a character to stdout	XENIX	L90
fputs	Put string to Level 2 file	ANSI	L91
fqsort	Sort an array of floats	LATTICE	L166
fread	Read blocks from a Level 2 file	ANSI	L93
free	Free Level 3 memory	ANSI	L19
freopen	Reopen a Level 2 file	ANSI	L95
frexp	Split fraction and exponent	ANSI	L96
fscanf	Formatted input from a file	ANSI	L97
fseek	Set Level 2 file position	ANSI	L102
ftell	Get Level 2 file position	ANSI	L102
fwrite	Write blocks to a Level 2 file	ANSI	L93
gcvt	Convert float to string	UNIX	L104
geta4	Establish addressability to the global data area	AmigaDOS	L106
getasn	Get assigned device	ANSI	L107
getc	Get a character from a file	ANSI	L63
getcd	Get current directory	AmigaDOS	L109
getcd	Get current directory	AmigaDOS	L110
getcd	Get current directory	AmigaDOS	L110

NAME	DESCRIPTION	CLASS	PAGE
getchar	Get a character from stdin	ANSI	L63
getcwd	Get current working directory	UNIX	L112
getdfs	Get free disk space	AmigaDOS	L113
getenv	Get environment variable	ANSI	L115
getfa	Get file attribute	AmigaDOS	L117
getfnl	Get filename list	LATTICE	L118
getft	Get file time	AmigaDOS	L121
getmem	Get Level 2 memory block (short)	LATTICE	L122
getml	Get Level 2 memory block (long)	LATTICE	L122
getpath	Get the path for a specific directory/file	AmigaDOS	L124
getreg	Obtain 68000-specific registers	LATTICE	L125
gets	Get string from stdin	ANSI	L64
GfxBase	Graphics Library Vector	AmigaDOS	L126
gmtime	Unpack Greenwich Mean Time (GMT)	ANSI	L127
iabs	Integer absolute value	LATTICE	L2
IntuitionBase	Intuition Library Vector	AmigaDOS	L129
iomode	Change mode of Level 1 file	LATTICE	L130
isalnum	Test if alphanumeric character	ANSI	L131
isalpha	Test if alphabetic character	ANSI	L131
isascii	Test if ASCII character	LATTICE	L131
iscntrl	Test if control character	ANSI	L131
iscsym	Test if C symbol character	LATTICE	L131
iscsymf	Test if C symbol lead character	LATTICE	L131
isdigit	Test if decimal digit character	ANSI	L131
isgraph	Test if graphic character	ANSI	L131
islower	Test if lower case character	ANSI	L131
isprint	Test if printable character	ANSI	L131
ispunct	Test if punctuation character	ANSI	L131
isspace	Test if space character	ANSI	L131
isupper	Test if upper case character	ANSI	L131
isxdigit	Test if hex digit character	ANSI	L131
jrand48	Random long (external seed)	UNIX	L44
labs	Long integer absolute value	XENIX	L2
lcong48	Set linear congruence parameters	UNIX	L44
ldexp	Load exponent	ANSI	L96

NAME	DESCRIPTION	CLASS	PAGE
localtime	Unpack local time	ANSI	L127
log	Natural logarithm function	ANSI	L57
log10	Base 10 logarithm function	ANSI	L57
longjmp	Perform long jump	ANSI	L135
lqsort	Sort an array of long integers	LATTICE	L166
lrand48	Random positive long (internal seed)	UNIX	L44
lsbrk	Allocate Level 1 memory (long)	LATTICE	L136
lseek	Set Level 1 file position	UNIX	L138
main	Your main or principal function	ANSI	L141
malloc	Allocate Level 3 memory	ANSI	L19
MathBase	FFP Library Vector	AmigaDOS	L145
matherr	Math error handler	UNIX	L146
MathTranBase	FFP Library Vector	AmigaDOS	L250
max	Compute maximum of two values	UNIX	L149
memccpy	Copy a memory block up to a char	ANSI	L151
memchr	Find a character in a memory block	ANSI	L151
MemCleanup	Deallocate all allocated memory	AmigaDOS	L150
memcmp	Compare two memory blocks	ANSI	L151
memcpy	Copy a memory block	ANSI	L151
memset	Set a memory block to a value	ANSI	L151
min	Compute minimum of two values	UNIX	L149
mkdir	Make a new directory	UNIX	L154
modf	Split floating point value	ANSI	L68
movmem	Move a memory block	UNIX	L151
mrand48	Random long (internal seed)	UNIX	L44
msflag	MS-DOS File Pattern Flag	LATTICE	L155
nrand48	Random positive long (external seed)	UNIX	L44
onbreak	Plant break trap	AmigaDOS	L156
onexit	Set an exit trap	ANSI	L158
open	Open a Level 1 file	UNIX	L160
perror	Print UNIX error message	ANSI	L162
poserr	Print AmigaDOS error message	AmigaDOS	L163
pow	Power function	ANSI	L57
pow2	Compute 2**x	AmigaDOS	L57
printf	Formatted printf to stdout	ANSI	L83

NAME	DESCRIPTION	CLASS	PAGE
putc	Put a character to a level 2 file	ANSI	L90
putchar	Put a character to stdout	ANSI	L90
putenv	Put string into environment	UNIX	L164
putreg	Set up 68000-specific registers	LATTICE	L125
puts	Put string to stdout	ANSI	L91
qsort	Sort a data array	UNIX	L166
rand	Generate a random number	ANSI	L168
rbrk	Release Level 1 memory	UNIX	L136
read	Read from Level 1 file	UNIX	L170
realloc	Re-allocate Level 3 memory	ANSI	L19
remove	Remove a file	ANSI	L172
rename	Rename a file	ANSI	L174
repmem	Replicate values through a block	UNIX	L151
rewind	Seek to beginning of Level 2 file	ANSI	L102
rlsmem	Release a Level 2 memory block	LATTICE	L176
rlsml	Release a Level 2 memory block	LATTICE	L176
rmdir	Remove a directory	UNIX	L179
rstmem	Reset memory pool	OLD	L180
sbrk	Allocate Level 1 memory (unsigned)	UNIX	L136
scanf	Formatted input from stdin	ANSI	L97
seed48	Set all 48 bits of internal seed	UNIX	L44
setbuf	Set buffer mode for a Level 2 file	ANSI	L181
setjmp	Set long jump parameters	ANSI	L135
setmem	Set a memory block to a value	UNIX	L151
setnbf	Set non-buffer mode for L2 file	UNIX	L181
setvbuf	Set variable buffer for L2 file	ANSI	L181
signal	Establish event traps	ANSI	L183
sin	Sine function	ANSI	L251
sinh	Hyperbolic sine function	ANSI	L251
sizmem	Get Level 2 memory pool size	LATTICE	L185
sprintf	Formatted print to a string	ANSI	L83
sqrt	Square root function	ANSI	L57
sqsort	Sort an array of short integers	LATTICE	L166
srand	Set seed for rand function	ANSI	L168
srand48	Set high 32 bits of internal seed	UNIX	L44

NAME	DESCRIPTION	CLASS	PAGE
sscanf	Formatted input from a string	ANSI	L97
stcarg	Get an argument	LATTICE	L186
stccpy	Copy one string to another	ANSI	L188
stcd_i	Convert decimal string to int	LATTICE	L190
stcd_l	Convert decimal string to long int	LATTICE	L190
stcgfe	Get file extension	LATTICE	L192
stcgfn	Get file node	LATTICE	L192
stcgfp	Get file path	LATTICE	L192
stch_i	Convert hexadecimal string to int	LATTICE	L190
stch_l	Convert hexadecimal string to long	LATTICE	L190
stci_d	Convert int to decimal	LATTICE	L196
stci_h	Convert int to hexadecimal	LATTICE	L196
stci_o	Convert int to octal	LATTICE	L196
stcis	Measure span of chars in set	LATTICE	L194
stcisn	Measure span of chars not in set	LATTICE	L194
stcl_d	Convert long int to decimal	LATTICE	L196
stcl_h	Convert long int to hexadecimal	LATTICE	L196
stcl_o	Convert long int to octal	LATTICE	L196
stclen	Measure length of a string	LATTICE	L225
stco_i	Convert octal string to int	LATTICE	L190
stco_l	Convert octal string to long int	LATTICE	L190
stcpm	Un-anchored pattern match	LATTICE	L199
stcpma	Anchored patter match	LATTICE	L199
stcsma	UNIX string pattern match (anchored)	AmigaDOS	L13
stcu_d	Convert unsigned int to decimal	LATTICE	L196
stcul_d	Convert unsigned long to decimal	LATTICE	L196
stpblk	Skip blanks (white space)	LATTICE	L201
stpbrk	Find break character in string	LATTICE	L203
stpchr	Find character in string	LATTICE	L205
stpchrn	Find character not in string	LATTICE	L205
stpcpy	Copy one string to another	ANSI	L188
stpdate	Convert date array to string	LATTICE	L207
stpsym	Get next symbol from a string	LATTICE	L209
stptime	Convert time array to string	LATTICE	L211
stptok	Get next token from a string	LATTICE	L213

NAME	DESCRIPTION	CLASS	PAGE
strbpl	Build string pointer list	LATTICE	L215
strcat	Concatenate strings	ANSI	L217
strchr	Find character in string	ANSI	L205
strcmp	Compare strings	ANSI	L219
strcmpi	Compare strings, case-insensitive	ANSI	L219
strcpy	Copy one string to another	ANSI	L188
strcspn	Measure span of chars not in set	ANSI	L194
strdup	Duplicate a string	XENIX	L222
stricmp	Compare strings, case-insensitive	ANSI	L219
strins	Insert a string	LATTICE	L223
strlen	Measure length of a string	ANSI	L225
strlwr	Convert string to lower case	XENIX	L226
strmfe	Make file name with extension	LATTICE	L227
strmfn	Make file name from components	LATTICE	L228
strmfp	Make file name from path/node	LATTICE	L230
strmid	Return a substring from a string	AmigaDOS	L231
strncat	Concatenate strings, max length	ANSI	L217
strncmp	Compare strings, length-limited	ANSI	L219
strncpy	Copy string, length-limited	ANSI	L188
strnicmp	Compare strings, no case, max size	ANSI	L219
strnset	Set string to value, max length	XENIX	L232
strpbrk	Find break character in string	ANSI	L203
strrchr	Find character not in string	ANSI	L205
strrev	Reverse a character string	XENIX	L233
strset	Set string to value	XENIX	L232
strsfn	Split the file name	LATTICE	L234
strspn	Measure span of chars in set	ANSI	L194
strsrt	Sort string pointer list	LATTICE	L237
strtok	Get a token	ANSI	L239
strtol	Convert string to long integer	UNIX	L241
strupr	Convert string to upper case	XENIX	L226
stspfp	Parse file path	LATTICE	L243
stub	Default routine for undefined routines	AmigaDOS	L244
swmem	Swap two memory blocks	UNIX	L151
sys_errlist	UNIX error messages	UNIX	L52

NAME	DESCRIPTION	CLASS	PAGE
sys_nerr	Number of UNIX error codes	UNIX	L52
system	Call system command processor	ANSI	L245
tan	Tangent function	ANSI	L251
tanh	Hyperbolic tangent function	ANSI	L251
tell	Get Level 1 file position	UNIX	L138
time	Get system time in seconds	ANSI	L246
timer	Get system clock with microseconds	AmigaDOS	L247
timezone	Timezone bias from GMT	UNIX	L269
toascii	Convert character to ASCII	LATTICE	L248
tolower	Convert character to lower case	ANSI	L248
toupper	Convert character to upper case	ANSI	L248
tqsort	Sort an array of text pointers	LATTICE	L166
tzdtn	Daylight time name	LATTICE	L269
tzname	Timezone names	UNIX	L269
tzset	Set time zone variable	XENIX	L253
tzstn	Standard time name	LATTICE	L269
ungetc	Push input character back	ANSI	L255
unlink	Remove a file	UNIX	L172
utpack	Pack UNIX time	LATTICE	L257
utunpk	Unpack UNIX time	LATTICE	L257
wait	Wait for child process to complete	UNIX	L76
waitm	Wait for multiple child processes	UNIX	L76
write	Write to Level 1 file	UNIX	L170
xcexit	Terminate with closing files	ANSI	L55

NAME

abort Abort the current process

SYNOPSIS

```
#include <stdlib.h>

abort();
```

DESCRIPTION

This function aborts the current process and returns a completion code of 3 to the parent process. Also the message "Abnormal program termination" is sent to *stderr*. Level 2 I/O buffers are not flushed.

RETURNS

The function does not return.

SEE

exit, _exit

NAME

abs	Absolute value
fabs	Float/double absolute value
iabs	Integer absolute value
labs	Long integer absolute value

SYNOPSIS

```
#include <math.h>

ax = abs(x);
ad = fabs(d);
ai = iabs(i);
al = labs(l);

double ad,d;
int ai,i;
long al,l;
```

DESCRIPTION

These functions compute the absolute value of the various numeric data types.

The most general approach is to use the *abs* macro, which is defined in the **math.h** header file. This macro accepts any data type as its argument and generates in-line code to perform the conversion. The definition is:

```
#define abs(x) ((x)<0?-(x):(x))
```

To minimize code size, you can use one of the function calls listed above instead of the *abs* macro. However, you must be careful to choose the function that corresponds to the data type being converted. Note that *fabs* works with either a float or a double as its argument because float arguments are automatically promoted to double. Similarly, *iabs* will work with ints or shorts.

Note that *abs* has a built-in version which is functionally equivalent to the standard library version. The statement *#include <string.h>* provides a default setting by which built-in functions are accessed. If you don't want the built-in function, you can use an #undef statement as follows: #undef abs.

NAME

access Check file accessibility

SYNOPSIS

```
#include <stdio.h>

ret = access(name,mode);

int ret;        return code
char *name;     file name
int mode;       access mode
```

DESCRIPTION

This function checks if a file is accessible in the way specified by *mode*, which follows the UNIX format:

RETURN		MEANING
0	=>	Check if file exists
1	=>	Check if file is executable
2	=>	Check if file is writable
3	=>	Check if file is writable and executable
4	=>	Check if file is readable
5	=>	Check if file is readable and executable
6	=>	Check if file is readable and writable
7	=>	Check if file is readable, writable, and executable

RETURNS

A return value of 0 indicates that access is allowed. If access is denied or the file cannot be found, -1 is returned. Additional error information can then be found in *errno* and *_OSERR*.

SEE

chmod, errno, _OSERR

NAME

argopt Get options from argument list

SYNOPSIS

```
#include <stdlib.h>

optd = argopt(argc,argv,opts,argn,optc);

char *optd;       option data pointer
int argc;         argument count
char *argv[];     argument vector
char *opts;       options expecting data
int *argn;        next argument number (changed)
char *optc;       option character (changed)
```

DESCRIPTION

This function examines an argument list to find the next option argument, using the conventions similar to those of the UNIX *shell* command processor. These conventions are:

1. n option is an argument that begins with a slash (/) or a dash (i.e. a minus sign) and appears between the command verb (i.e. argv[0]) and the first non-option argument. The reason we recognize either a slash or a dash is that the former is a convention used by some AmigaDOS commands, while the latter has been used by UNIX for a long time.

2. The character immediately following the dash is called the *option character*, and it may be followed by a character string known as the *option data.*

3. If the option character appears in the *opts* string, then the data can be separated from the character by white space. In effect, this means that the data might be in the next *argv* entry if it does not follow the option character in the current entry.

4. A dash or slash followed by a blank or a dash indicates the end of the options.

Each time argopt is called, it will find the next option in the argument array and update the integer referenced by *argn*. On the first call, you should set this integer to 1, since *argv[0]* points to the command verb. The *argc* and *argv* items are normally the same as those passed to your main program, and they are not changed as a result of the *argopt* calls. The option character is returned in the byte referenced by *optc*, and the function returns a pointer to the option data string or to a null byte. If the next entry in *argv* is not an option, then the function returns a NULL pointer.

The *opts* item provides some flexibility in the way the option data is handled. If *opts* points to an empty string, then any option data must immediately follow the option character. However, if *opts* is not empty, then it lists the option characters that always have data. For those characters, the data can be preceded by white space in the command line. What this actually means is that *argopt* will look at the next entry in *argv* if the option character is not followed by a data string. If that next entry does not begin with a dash, then it is taken as the option data. See the examples below for clarification.

RETURNS

If the next argument is not an option, the function returns a NULL pointer. Otherwise, it returns a pointer to the option data, which will be an empty string if there was no data. Also, if an option was found, the character is placed into the byte referenced by *optc*, and *argn* is adjusted to index the next entry in *argv*.

SEE

main

EXAMPLE

```
/*
*
* Assume that this program is invoked by the following
* command line:
*
*       myprog -x -ypdq -z -g moo -g - blah
*
* The output will then be:
*
*       Option: x Data:
*       Option: y Data: pdq
*       Option: z Data:
*       Option: g Data: moo
*       Option: g Data:
*       Arg[8]: blah
*
*/
#include <stdlib.h>
char opts[ ] = "gx";
main(argc,argv)
int argc;
char *argv[ ];
{
char option,*odata;
int next;
next = 1;
for(; (odata = argopt(argc,argv,opts,&next,&option))
     != NULL;)
printf("Option: %c, Data: %s\n",option,odata);

for(; next < argc; next++)
        printf("Arg[%d]: %s\n",next,argv[next]);
}
```

NAME

asctime Generate ASCII time string

SYNOPSIS

```
#include <time.h>

s = asctime(t);

char *s;          points to time string
struct tm *t;     points to time structure
```

DESCRIPTION

This function converts a time structure into an ASCII string of exactly 26 characters having the form:

```
"DDD MMM dd hh:mm:ss YYYY\n\0"
```

where **DDD** is the day of the week, **MMM** is the month, **dd** is the day of the month, **hh:mm:ss** is the hour:minute:seconds, and **YYYY** is the year. An example is:

```
"Wed Sep 04 15:13:22 1985\n\0"
```

The time pointer returned by the function refers to a static data area that is shared by both *ctime* or *asctime*. The time structure argument *t* is usually returned by the *gmtime* or *localtime* function.

SEE

gmtime, localtime

EXAMPLE

```
#include <time.h>
#include <stdio.h>

main()
{
struct tm *tp;
long t;

time(&t);
tp = localtime(&t);
printf("Current time is %s",asctime(tp));
}
```

NAME

assert	Assert program validity
_assert	Failure exit for assert

SYNOPSIS

```
#include <assert.h>

assert(x);
_assert(exp,file,line);

char *exp;        failing expression
char *file;       source file name
char *line;       source line number
```

DESCRIPTION

The *assert* macro tests an expression **x** for validity (non-zero value). If the expression is 0, then the macro calls the *_assert* function with the expression in text form plus the source file name and line number, also as text strings. The default version of *_assert* prints a message on *stderr* and aborts with an exit code of 1. You can replace this function with one of your own if you require some other action when an assertion fails. The source code is supplied in the compiler package.

Note that the **assert.h** header file must be included in your program in order to define the macro. Also, the file contains two versions of the macro. If the symbol **NDEBUG** is defined, then a null version of the macro is used; otherwise the normal code-generating version applies. This allows you to strip the assertion code from your program without removing the *assert* calls. To do this, define **NDEBUG** in one of your header files or on the compiler command line via the **-d** option. In the former case, the header file containing the **NDEBUG** definition must be included before **assert.h**.

EXAMPLE

```
#include <assert.h>

/* Make sure integer x is positive */

postest(x)
int x;
{
assert(x >= 0);
}
```

NAME

astcsma	AmigaDOS string pattern match (anchored)
stcsma	UNIX string pattern match (anchored)

SYNOPSIS

```
#include <string.h>

length = astcsma (s,p);
length = stcsma (s,p);

int length;         /* length of matching string */
char *s;            /* string being scanned */
char *p;            /* pattern string */
```

DESCRIPTION

These functions perform a simple anchored pattern match. The pattern must match at the beginning of the supplied string.

The function *astcsma* performs a simple pattern match of the type used by AmigaDOS.

The function *stcsma* performs a simple pattern match of the type used by the UNIX shell. The only meta-characters recognized are "*" and "?". The "*" matches an arbitrary number of characters, and the "?" matches exactly one character.

RETURNS

Both functions return the length of the matching string or zero if there was no match.

NAME

atexit Set an exit trap

SYNOPSIS

```
#include <stdlib.h>

error = atexit(func);

int error;              /* non-zero for success */
int (*func)();          /* trap function pointer */
```

DESCRIPTION

This function plants an exit trap, which will be called when the program exits. This is an alternate entry point for the function *onexit*. See the description of *onexit* for complete details.

RETURNS

A zero is returned for success and a non-zero value is returned if an error is encountered.

SEE

onexit

NAME

atof Convert ASCII to float

SYNOPSIS

```
#include <math.h>

d = atof(p);

double d;      floating point result
char *p;       input string pointer
```

DESCRIPTION

This function converts an ASCII input string into a double value. The string can contain leading white space and a plus or minus sign, followed by a valid floating point number in normal or scientific notation. If scientific notation is used, there can be no white space between the number and the exponent. For example,

```
123.456e-53
```

is a valid number in scientific notation.

EXAMPLE

```
/*
*
* This program tests the atof function.
*
*/
#include <stdio.h>
#include <math.h>

main()
{
char buff[80];
double d;
```

```
while(1)
        {
        printf("\nEnter a number: ");
        if(gets(buff) == NULL) exit(0);
        if(buff[0] == '\0') exit(0);
        d = atof(buff);
        printf("%e\n",d);
        }
}
```

NAME

atoi	Convert ASCII to integer
atol	Convert ASCII to long integer

SYNOPSIS

```
#include <stdlib.h>

x = atoi(s);
y = atol(s);

int x;
long y;
char *s;
```

DESCRIPTION

These functions convert ASCII strings into normal or long integers. The string must have the form:

```
[whitespace][sign]digits
```

where **[whitespace]** indicates optional leading white space, **[sign]** indicates an optional + or - sign character, and **digits** is a contiguous string of digit characters. Once the digit portion is reached, the conversion continues until a non-digit character is hit. No check is made for integer overflow.

RETURNS

As noted above.

SEE

atof, stcd_i, stcd_l

NAME

bldmem Build a memory pool of specified size

SYNOPSIS

```
#include <stdlib.h>

bldmem(n);
int n;          /* number of 1K-byte blocks in pool */
```

DESCRIPTION

The *bldmem* function uses *sbrk* to get up to n contiguous 1K-byte blocks of memory for the pool. If n is 0, the pool is initialized but no memory is allocated.

RETURNS

Returns a -1 if first *sbrk* fails.

SEE

getmem, getml, rlsmem, rlsml, sizmem, sbrk

NAME

calloc	Allocate and clear Level 3 memory
free	Free Level 3 memory
malloc	Allocate Level 3 memory
realloc	Re-allocate Level 3 memory

SYNOPSIS

```
#include <stdlib.h>

b = calloc(nelt,esize);
b = malloc(n);
nb = realloc(b,n);
error = free(b);

char *b;            block pointer
unsigned nelt;      number of elements
unsigned esize;     element size
unsigned n;         number of bytes
char *nb;           new block pointer
int error;          0 for success, -1 for failure
```

DESCRIPTION

These functions form Level 3 of Lattice's layered memory allocation system. This level is fully compatible with UNIX and with the ANSI standard.

The *malloc* function allocates a block that is *n* bytes long and is aligned in such a way that you can cast the block pointer to any pointer type. If the block cannot be allocated, a NULL pointer is returned. The *calloc* function uses *malloc* to get a block whose size in bytes is given by:

```
n = nelts * esize;
```

Then the block is cleared to zeroes. In common with *malloc, calloc* returns a NULL pointer if the block cannot be allocated.

A call to *realloc* will obtain a new block whose size is **n** bytes. Then it copies

the old block **b** to the new block **nb** and releases the old block. If it is less than the old block size, only the first **n** bytes are copied. As described below, you can, under certain circumstances, re-allocate a block that has been freed.

By ANSI definition, *calloc*, *malloc*, and *realloc* can only allocate 64K at a time.

The *free* function releases a block that was previously obtained via *calloc*, *malloc* or *realloc*. For compatibility with some versions of UNIX, the block is not actually returned to the free space pool until the next time you call *calloc*, *malloc*, *realloc* or *free*. Then if that next call is to *realloc* and the block being re-allocated is the one that was just freed, *realloc* will proceed correctly. In other words, you can ask *realloc* to re-allocate a block that was freed as long as you have not called *calloc*, *malloc* or *realloc* in the meantime.

RETURNS

For *calloc*, *malloc* and *realloc*, a NULL pointer is returned if there is not enough space for the requested block. For *free*, 0 is returned if the block was successfully released; otherwise, the function returns -1.

> NOTE: These functions are called *Level 3 memory allocation* because they call upon Level 2 functions *getmem* and *rlsmem*. You should be aware that the Level 1 function *rbrk* frees all Level 2 and Level 3 blocks. Be careful!

SEE

getmem, rlsmem, sbrk, rbrk

EXAMPLE

```
#include "stdio.h"
#include "stdlib.h"
#include "string.h"

struct LIST
{
```

```
        struct LIST *next;
        char text[2];
};

void main(argc, argv, envp)
int argc;
char *argv[];
char *envp[];
{
        struct LIST *p;
        struct LIST *q;
        struct LIST list;
        char b[256];
        int x;

        while (1)
        {
                printf("\nBegin new group...\n");
                for (q = &list; ; q = p)
                {
                        printf("Enter a text string: ");
                        if (gets(b) == NULL)
                                break;
                        if (b[0] == NULL)
                        {
                                if (q == &list)
                                        exit(0);
                                break;
                        }
                        x = sizeof(struct LIST) - 2 +
                            strlen(b) + 1;
                        p = (struct LIST *)malloc(x);
                        if (p == NULL)
                        {
                                printf("No more memory\n");
                                break;
                        }
                        q->next = p;
                        p->next = NULL;
                        strcpy(p->text, b);
                }
                printf("\n\nTEXT LIST...\n");
```

```
                for (p = list.next; p != NULL; p = p->next)
                {
                        printf("%s\n", p->text);
                        free((char *)p);
                }
                list.next = NULL;
        }

        return;
}
```

NAME

ceil	Get ceiling of a real number
floor	Get floor of a real number

SYNOPSIS

```
#include <math.h>

x = ceil(y);
x = floor(y);

double x,y;
```

DESCRIPTION

These functions return the integral values that are nearest to the specified real number. For *ceil*, the return is the next higher integer, while *floor* returns the next lower integer.

> NOTE: Even though these functions return integral values, the results are still real numbers.

EXAMPLE

```
#include <math.h>

double r;

r = ceil(523.96);     /* r contains 524.0 */
r = floor(523.96);    /* r contains 523.0 */
```

NAME

chdir Change current directory

SYNOPSIS

```
#include <stdio.h>

error = chdir(path);
int error;      0 if successful
char *path;     points to new directory path string
```

DESCRIPTION

This function changes the current directory to the specified path. For AmigaDOS, the path may begin with a drive or volume name and a colon.

RETURNS

If the return value is non-zero, then the operation failed. An AmigaDOS error code will be in *_OSERR*, and a UNIX error code will be in *errno*.

SEE

mkdir, rmdir, getcd, getcwd, errno, _OSERR

NAME

chkabort Check for Break character

SYNOPSIS

```
void chkabort();
```

DESCRIPTION

This function forces immediate checking for the break characters Ctrl-C and Ctrl-D. Normally this check only occurs when level 1 I/O is performed. The *chkabort* function provides a mechanism for detecting the break characters at other times. This can be important in programs which do little or no I/O processing. (Note that *chkabort* ignores the break characters Ctrl-E and Ctrl-F.)

If either break key is detected, the break trap is executed. The break trap is simply a function that gets called whenever the user keys Ctrl-C or Ctrl-D. If the break trap returns a zero value, control returns to the calling function. Otherwise the program terminates. If the program is invoked from the Workbench, the default handler displays a requester allowing the user to abort the program. Otherwise, the default handler will print a message to the CLI and abort.

Note that *chkabort* may be replaced via calls to *onbreak* or *signal*. Also, an alias for *chkabort( )* is Chk Abort.

RETURNS

There is no return value.

SEE

signal, onbreak

NAME

chkml Check for largest memory block

SYNOPSIS

```
#include <stdlib.h>

size = chkml();
long size;
```

DESCRIPTION

This function returns the size, in bytes, of the largest block that is currently available in the level 2 memory pool without calling upon the operating system to supply additional heap space.

Note: the return value is a long integer, so the function must be declared appropriately.

SEE

getmem, getml, rlsmem, rlsml, sizmem

NAME

chkufb Check Level 1 file handle

SYNOPSIS

```
#include <ios1.h>

ufb = chkufb(fh);

struct UFB *ufb;    pointer to UNIX file block
int fh;             file handle
```

DESCRIPTION

This function checks if an AmigaDOS file handle is currently associated with a Level 1 file. Normally it is used internally by *open*, *close*, *read*, *write*, *lseek* and *tell*. The UFB structure is defined in header file **ios1.h**. For AmigaDOS this structure is two integers. The first contains the mode flags specified in the call to the *open* function. The second long integer contains the file handle. The external name *_ufbs* refers to an array of UFB structures, and the external integer *_nufbs* indicates how many structures are in the array. Normally this value is twenty.

RETURNS

If no UFB is currently attached to the file handle, a null pointer is returned.

NAME

chmod Change file protection mode

SYNOPSIS

```
#include <stdio.h>
#include <fcntl.h>

error = chmod(name,mode);

int error;      error code
char *name;     file name
int mode;       protection mode
```

DESCRIPTION

This function changes a file's protection mode. It is compatible with UNIX, although AmigaDOS also provides separate delete protection for each file. The **mode** argument should be formed by ORing any combination of the following symbols which are defined in **fcntl.h:**

Value	Meaning
S_IWRITE	Write permission
S_IREAD	Read permission
S_IEXECUTE	Execute permission
S_IDELETE	Delete permission
S_IPURE	Set the bit
S_ARCHIVE	Indicate that a file has been backed up

RETURNS

If the operation is successful, the function returns 0. Otherwise it returns -1 and places error information in *errno* and *_OSERR*.

SEE

access, errno, _OSERR

EXAMPLE

```
/*
*
* This example changes file "xyz/pdq.x" so it
* can be read and written.
*
*/
#include <stdlib.h>
#include <fcntl.h>

if(chmod("xyz/pdq.x",S_IWRITE | S_IREAD))
  perror("Change mode");
```

NAME

clearerr	Clear Level 2 I/O error flag
clrerr	Clear Level 2 I/O error flag

SYNOPSIS

```
#include <stdio.h>

clrerr(fp);
clearerr(fp);

FILE *fp;           file pointer
```

DESCRIPTION

These functions clear the error flag associated with the specified Level 2 file. Once set, the error flag forces an EOF return any time the file is accessed until the flag is reset.

Note that *clearerr* is a macro, while *clrerr* is a function. The former is provided for compatibility with some older versions of UNIX.

RETURNS

None.

SEE

fopen

NAME

close Close a Level 1 file

SYNOPSIS

```
#include <fcntl.h>

error = close(fh);

int error;      non-zero if error
int fh;         file handle
```

DESCRIPTION

This function closes a Level 1 file that was previously opened via the *open* function. Any pending output is completed and the file directory is updated.

All Level 1 files are automatically closed when your program terminates, but it is good programming practice to close a file when you are finished with it. One reason for doing this is to free up the operating system resources (e.g. control blocks and buffers) that are allocated for the file while it remains open.

RETURNS

The function returns 0 if it is successful. Otherwise, it returns -1 and places additional error information into *errno* and *_OSERR*.

SEE

errno, open, _OSERR

EXAMPLE

See the *open* function.

NAME

creat Create a Level 1 file

SYNOPSIS

```
#include <fnctl.h>

fh = creat(name,prot);

int fh;         file handle
char *name;     file name
int prot;       protection mode
```

DESCRIPTION

This function is exactly the same as calling the *open* function in the following way:

```
open(name,O_WRONLY | O_TRUNC | O_CREAT , prot)
```

In other words, the file is created if it doesn't exist and truncated if it does exist. Then it is opened for writing. The protection mode can be any of the following:

Value	Meaning
S_IWRITE	Write permission
S_IREAD	Read permission
S_IWRITE | S_IREAD	Write and read permission
0	Write and read permission

In the current AmigaDOS implementation the protection mode is ignored.

RETURNS

If the operation is successful, the function returns a file handle, which is an integer equal to or greater than 0. Otherwise it returns -1 and places error information in *errno* and *_OSERR*.

SEE

errno, _OSERR, chgfa, chmod, close, open

NAME

ctime Convert time value to string

SYNOPSIS

```
#include <time.h>

s = ctime(t);
char *s;      points to time string
long *t;      points to time value
```

DESCRIPTION

This function converts a Greenwich Mean Time (GMT) time value to an ASCII string of exactly 26 characters having the form:

```
"DDD MMM dd hh:mm:ss YYYY\n\0"
```

where

DDD is the day of the week, **MMM** is the month, **dd** is the day of the month, **hh:mm:ss** is the hour:minute:seconds and **YYYY** is the year. An example is:

```
"Wed Sep 04 15:13:22 1985\n\0"
```

The time pointer returned by the function refers to a static data area that is shared by both *ctime* or *asctime*.

The time value argument **t** must point to a long integer that is the number of seconds since 00:00:00 Greenwich Mean Time, January 1, 1970. Normally this value is obtained from the **time** function. Note that *ctime* converts this value back into local time by calling *tzset* and then subtracting the contents of *timezone*.

NOTE: **t** is a pointer to a long integer. A common error is to pass the integer itself instead of the pointer. Observe the use of the ampersand (&) operator in the following example.

SEE

asctime, gmtime, localtime, time

EXAMPLE

```
#include <time.h>
#include <stdio.h>

main()
{
long t;

time(&t);
printf("Current time is %s",ctime(&t));
}
```

NAME

CXFERR Low-level float error

SYNOPSIS

```
#include &lt;math.h&gt;

CXFERR(code);
int code;
```

DESCRIPTION

This function is called when an error is detected by one of the low-level floating point routines, such as the arithmetic operations. Higher-level routines, such as the trigonometric functions, use the more sophisticated *matherr* function.

Programmers can replace this error trap with an application-dependent routine, as long as they still store the error code into the global integer *_FPERR*. This is necessary because some of the math functions check *_FPERR* to see if low-level errors occurred.

The error code passed to *CXFERR* indicates the type of floating point anomaly that occurred, as follows:

Symbol	Value	Meaning
FPEUND	1	Underflow
FPEOVF	2	Overflow
FPEZDV	3	Divide by zero
FPENAN	4	Not a number
FPECOM	5	Not comparable

These codes are defined in **math.h**.

SEE

_FPERR, _FPA, matherr

NAME

datecmp Compare two AmigaDOS dates

SYNOPSIS

```
#include <dos.h>

status = datecmp(d1, d2);

int status;               /* result of comparison:
                                0  if equal
                                -1 if d1 > d2
                                1  if d1 < d2  */
struct DateStamp *d1;     /* first date to compare */
struct DateStamp *d2;     /* second string to compare */
```

DESCRIPTION

The *datecmp* function compares two date vectors (three longwords) to see if they are equal. To be equal, all of days, minutes, and ticks must match. If unequal, an appropriate return is sent back.

RETURNS

If the dates match as described above, then 0 is returned. If the first date is greater than the second date, then -1 is returned. Otherwise, 1 is returned.

SEE

DateStamp (AmigaDOS Technical Reference)

NAME

_dclose Close an AmigaDOS file

SYNOPSIS

```
#include <dos.h>

error = _dclose(fh);

int error;    0 for success, -1 for error
int fh;       file handle
```

DESCRIPTION

This function closes an AmigaDOS file that was opened via *_dcreat*, *_dcreatx* or *_dopen*. AmigaDOS will not automatically close these files for you. One reason for doing this is to free up the operating system resources (e.g. control blocks and buffers) which are allocated for the file while it remains open.

If your program terminates normally, the run time support library will automatically close all files opened via level 1 or level 2 I/O calls.

RETURNS

If the operation is successful, the function returns 0. Otherwise it returns -1 and places error information in *errno* and *_OSERR*.

SEE

errno, _OSERR, _dcreat, _dcreatx, _dopen, _dread, _dwrite, _dseek

NAME

_dcreat	Create or truncate AmigaDOS file
_dcreatx	Create new AmigaDOS file

SYNOPSIS

```
#include <dos.h>

fh = _dcreat(name,fatt);
fh = _dcreatx(name,fatt);

int fh;          file handle (-1 for error)
char *name;      file name
int fatt;        file attribute
```

DESCRIPTION

These functions create and open an AmigaDOS file, returning the file handle. The *_dcreat* operation will truncate the file if it already exists, or create the file if it does not exist. Alternatively, *_dcreatx* will fail if the file already exists.

The file attribute argument is accepted to provide source level compatibility with other systems, but is ignored under AmigaDOS.

RETURNS

If the operation is successful, the function returns a file handle. Otherwise it returns -1 and places error information in *errno* and *_OSERR*.

SEE

errno, _OSERR, _dopen, _dclose, _dread, _dwrite, _dseek

NAME

dfind	Find first directory entry
dnext	Find next directory entry

SYNOPSIS

```
#include <dos.h>

error = dfind(info,name,attr);
error = dnext(info);

int error;                   0 if successful
struct FILEINFO *info;       file information area
char *name;                  file name or pattern
int attr;                    file attribute bits
```

DESCRIPTION

These functions search a directory for entries that match the specified file name or file name pattern. The *dfind* function locates the first matching file. Then successive calls to *dnext* locate additional matching files. Each *dnext* call must be given the file information that was returned on the preceding call to *dfind* or *dnext*.

The **name** argument must be a null-terminated string specifying the drive, path, and name of the desired file. The drive and path can be omitted, in which case the current directory will be searched. You can use the Amiga-DOS wildcard characters as defined in the AmigaDOS Users Manual for pattern matching in the name portion. For example, *xy#?.b* will locate files in the current directory that begin with *xy* and have *b* as their extension. Setting the external integer location **msflag** to a nonzero value will cause all subsequent calls to *dfind* and *dnext* to use the MS-DOS * and ? characters for pattern matching in the name portion.

The **attr** argument specifies which file types are to be included in the search. If **attr** is zero, the search will include only normal files. Otherwise the search will also include directories.

The **info** argument points to a file information structure as defined in the **dos.h** header file. For AmigaDOS, this is the same as the AmigaDOS FileInfoBlock structure:

```
struct FileInfoBlock {
   long   fib_DiskKey;
   long   fib_DirEntryType;
   char   fib_FileName[108];
   long   fib_Protection;
   long   fib_EntryType;
   long   fib_Size;
   long   fib_NumBlocks;
   struct DateStamp fib_Date;
   char   fib_Comment[116];
};
```

RETURNS

If the operation is successful, a value of 0 is returned. Otherwise, the return value is -1, and further error information can be found in *errno* and *_OSERR*.

> NOTE: **info** is a pointer to a file info block that must be allocated on a 4-byte (long word) boundary by the calling program. A common error is failing to allocate the structure before calling the function.

SEE

errno, _OSERR

NAME

_dopen Open an AmigaDOS file

SYNOPSIS

```
#include <dos.h>

fh = _dopen(name,mode);

int fh;         file handle (-1 for error)
char *name;     file name
int mode;       access mode
```

DESCRIPTION

This function opens an AmigaDOS file and returns the file handle. The **mode** argument must be a mode supported directly by AmigaDOS and defined in the Amiga header file **libraries/dos.h**.

RETURNS

If the operation is successful, the function returns a file handle. Otherwise it returns -1 and places error information in *errno* and *_OSERR*.

SEE

errno, _OSERR, open, _dcreat, _dcreatx, _dclose, _dread, _dwrite, _dseek

NAME

DOSBase AmigaDOS Library Vector

SYNOPSIS

```
extern long DOSBase;
DOSBase = OpenLibrary("dos.library",ver);
```

DESCRIPTION

This external location is used by various Amiga library routines which interface with AmigaDOS system functions. It is initialized by an **OpenLibrary** call in the startup code and is expected to contain the base address of the AmigaDOS system library vector table. If you do not link with the startup module **c.o** and you make calls to AmigaDOS system functions or use Lattice features that call AmigaDOS system functions, then you must first initialize this location by calling **OpenLibrary**.

NAME

drand48	Random double (internal seed)
erand48	Random double (external seed)
lrand48	Random positive long (internal seed)
nrand48	Random positive long (external seed)
mrand48	Random long (internal seed)
jrand48	Random long (external seed)
srand48	Set high 32 bits of internal seed
seed48	Set all 48 bits of internal seed
lcong48	Set linear congruence parameters

SYNOPSIS

```
#include <math.h>

x = drand48();
x = erand48(seed);

y = lrand48();
y = nrand48(seed);

z = mrand48();
z = jrand48(seed);

srand48(hseed);
pseed = seed48(seed);
lcong48(parm);

double x;          random double
long y;            random positive long
long z;            random long
short seed[3];     seed value (high bits in seed[0])
long hseed;        high 32 bits of seed value
short *pseed;      pointer to internal seed
short parm[7];     parameters
```

DESCRIPTION

These functions generate various types of random numbers using the linear congruential algorithm and 48-bit arithmetic. The normal functions *drand48, lrand48* and *mrand48* use an internal 48-bit storage area for the seed value. Special versions *erand48, jrand48* and *nrand48* are provided for cases where several seeds are in use at the same time, in which case the user specifies the seed on each function call.

The *drand48* and *erand48* functions return double values distributed uniformly over the interval from 0.0 up to but not including 1.0.

The *lrand48* and *nrand48* functions return non-negative long integers uniformly distributed over the interval from 0 to 2**31-1.

The *mrand48* and *jrand48* functions return signed long integers uniformly distributed over the interval from -2**31 to 2**31-1.

The *srand48* and *seed48* functions allow initialization of the internal 48-bit seed to something other than the default. For *srand48* the specified long value is copied into the high 32 bits of the seed, and the low 16 bits are set to 0x330E. For *seed48* the entire 48 bits are loaded from the specified array, and the function returns a pointer to the internal seed array.

The *lcong48* function allows a much more intricate initialization of the linear congruential algorithm. The algorithm is of the form:

```
X[n+1] = (a * X[n] + c) mod m
```

where **m** is 2**48 and the default values for **a** and **c** are 0x5DEECE66D and 0xB, respectively. The array passed to *lcong48* is structured as follows:

PARAMETER	VALUE
parm[0]	Bits 47-32 of value X[n]
parm[1]	Bits 31-16 of value X[n]
parm[2]	Bits 15-00 of value X[n]
parm[3]	Bits 47-32 of value a
parm[4]	Bits 31-16 of value a
parm[5]	Bits 15-00 of value a
parm[6]	value c

Whenever *seed48* is called, **a** and **c** are reset to their default values.

RETURNS

As noted above.

SEE

rand, srand

NAME

_dread	Read from an AmigaDOS file
_dwrite	Write to an AmigaDOS file

SYNOPSIS

```
#include <dos.h>

count = _dread(fh,buffer,length);
count = _dwrite(fh,buffer,length);

unsigned int count;         actual bytes read or written
int fh;                     file handle
char *buffer;               data buffer
unsigned int length;        number of bytes to read or write
```

DESCRIPTION

These functions read or write an AmigaDOS file whose handle was returned by *_dcreat*, *_dcreatx* or *_dopen*. Under normal circumstances, the value returned should match the buffer length. If this value is -1 or greater than the requested length, then some type of error occurred, and you should consult *errno* and *_OSERR*. If the actual length is less than the requested length when reading, this usually means that the file is exhausted. Similarly, if the actual length is less than the requested length for a write operation, this usually means that the device has no more space available. In both of these cases, it is still a good idea to check *errno* and *_OSERR* just in case some malfunction caused the short count.

RETURNS

If the operation is successful, the function returns the actual number of bytes transferred. Otherwise it returns -1 and places error information in *errno* and *_OSERR*.

SEE

errno, _OSERR, _dcreat, _dcreatx, _dopen, _dclose, _dseek

NAME

_dseek Re-position an AmigaDOS file

SYNOPSIS

```
#include <dos.h>

apos = _dseek(fh,rpos,mode);

long apos;   actual file position
int fh;      file handle
long rpos;   relative file position
int mode;    seek mode
```

DESCRIPTION

This function re-positions an AmigaDOS file whose handle was returned by *_dcreat*, *_dcreatx* or *_dopen*. The seek mode is the same as for *lseek* as follows:

Mode 0 Position is relative to beginning of file.

Mode 1 Position is relative to current file location.

Mode 2 Position is relative to end of file.

Note that for modes 1 and 2 **rpos** can be positive or negative, but **apos** is always the actual (positive) position relative to the beginning of file.

RETURNS

If the operation is successful, the function returns the actual file position, which is a long integer. Otherwise it returns -1L and places error information in *errno* and *_OSERR*.

SEE

errno, _OSERR, _dcreat, _dcreatx, _dopen, _dclose, _dread, _dwrite

NAME

ecvt	Convert float to string
fcvt	Convert float to string

SYNOPSIS

```
#include <math.h>

s = ecvt(v,dig,decx,sign);
s = fcvt(v,dec,decx,sign);

char *s;      string pointer
double v;     floating point value
int dig;      number of digits
int dec;      number of decimal places
int *decx;    pointer to decimal index (returned)
int *sign;    pointer to sign indicator
```

DESCRIPTION

These functions convert a floating point number into an ASCII character string consisting of digits only and terminated by a null character.

For *ecvt*, the second argument indicates the total number of digits that should be generated, while for *fcvt* it indicates how many digits should be generated to the right of the decimal place. If the floating point value contains fewer significant digits, zeroes are appended. If there are too many significant digits, the low order (right-most) digit is rounded.

The **decx** argument points to an integer that will receive a value indicating where the decimal point should be placed in the string. For example, an index value of 3 indicates that the decimal point should be placed just after the third character in the string. A value of zero means that the decimal point is just before the first character. If the index is negative, it indicates the number of zeroes that are between the decimal point and the first character. For example, -3 means that there are three zeroes between the decimal point and the beginning of the string.

The **sign** argument points to an integer that will be non-zero if **v** is negative.

EXAMPLE

```
#include <math.h>

main()
{
int decx,sign;
char *string;

string = ecvt(3.1415926535,10,&decx,&sign);

/*
*   string  =>  "3141592654"
*   decx    =>  1
*   sign    =>  0
*/

string = fcvt(3.1415926535,10,&decx,&sign);

/*
*   string  =>  "31415926535"
*   decx    =>  1
*   sign    =>  0
*/
}
```

NAME

emit Emit 68000 instruction word

SYNOPSIS

```
#include <dos.h>

emit 0x4180       Assembler instruction for chk d0,d0
```

DESCRIPTION

The built-in function *emit* takes a constant 16-bit value corresponding to a 68000 assembly language instruction and inserts it in-line with the code. However, it does not check whether the 16-bit value is a valid 680000 instruction. It lacks the power flexibility of an in-line assembler.

RETURNS

Returns a void.

SEE

getreg, putreg

> If one doesn't know how to use the *emit* function, it can create serious problems. While programmers may find this function useful in some situations, it should not be used without exercising a great deal of care and skill.

NAME

errno	UNIX error number
sys_nerr	Number of UNIX error codes
sys_errlist	UNIX error messages

SYNOPSIS

```
#include <error.h>
extern int errno;
extern int sys_nerr;
extern char *sys_errlist[ ];
```

DESCRIPTION

The external integer named *errno* is initialized to 0 at start-up time. Then if an error is detected by one of the standard library functions, a non-zero value is placed there. The standard library never resets *errno*.

Programmers typically use this information in two ways. In some cases, it is appropriate to check *errno* after a sequence of operations and abort if any error occurred along the way. In other cases, *errno* is checked periodically, and if it is non-zero, the appropriate corrective action is taken. Then the application program resets *errno* before beginning the next processing phase.

The **sys_nerr** and **sys_errlist** items are defined in a C source file named **syserr.c** and are used by the *perror* function to print messages that correspond to the code found in *errno*.

Note that even though error information is normally placed into *errno* by the standard library functions, application programs can also use this technique to indicate problems. However, you should be careful about adding new codes and messages just above the highest UNIX code currently defined, since new UNIX codes are occasionally added. Also, we recommend that you add application-dependent codes by extending the header file **error.h**, which contains symbolic definitions of the code numbers. The currently defined codes are listed below:

Class: UNIX

Symbol	Code	Meaning
EOSERR	-1	Operating system error
EPERM	01	User is not owner
ENOENT	02	No such file or directory
ESRCH	03	No such process
EINTR	04	Interrupted system call
EIO	05	I/O error
ENXIO	06	No such device or address
E2BIG	07	Arg list is too long
ENOEXEC	08	Exec format error
EBADF	09	Bad file number
ECHILD	10	No child process
EAGAIN	11	No more processes allowed
ENOMEM	12	No memory available
EACCES	13	Access denied
EFAULT	14	Bad address
ENOTBLK	15	Bulk device required
EBUSY	16	Resource is busy
EEXIST	17	File already exists
EXDEV	18	Cross-device link
ENODEV	19	No such device
ENOTDIR	20	Is not a directory
EISDIR	21	Is a directory
EINVAL	22	Invalid argument
ENFILE	23	No more files (system)
EMFILE	24	No more files (process)
ENOTTY	25	Not a terminal
ETXTBSY	26	Text file is busy
EFBIG	27	File is too large

ENOSPC	28	No space left
ESPIPE	29	Seek issued to pipe
EROFS	30	Read-only file system
EMLINK	31	Too many links
EPIPE	32	Broken pipe
EDOM	33	Math function argument error
ERANGE	34	Math function result is out of range

SEE

perror

NAME

exit	Terminate with clean-up
_exit	Terminate with no clean-up
xcexit	Terminate with closing files

SYNOPSIS

```
#include <stdlib.h>

exit(code);        int
_exit(code);
xcexit(lcode);     long lcode
```

DESCRIPTION

These functions terminate execution of the current program and return control to the parent program. Use *exit*, for a graceful termination, which means that all pending output buffers are written and all files are explicitly closed. The *_exit* function terminates immediately without writing output buffers or closing level 2 files. Generally, this latter form is used only in emergency situations when you do not care if some output data is lost.

In either case, files opened via *open*, *creat*, or *creatx* will be closed. However, files opened via *_dopen*, *_dcreat* or *_dcreatx* will not be closed.

The parameter **code** is a value from 0 to 255 that gets passed back to the parent. By convention, a value of zero indicates success. The *xcexit* function bypasses closing any files, but does call the standard termination routines.

RETURNS

This function does not return.

EXAMPLE

```
/*
*
* This example shows how you would abort a program
* if it is not called with a valid input file name.
*
*/
#include <stdlib.h>
#include <stdio.h>

main(argc,argv)
int argc;
char *argv[];
{
FILE *f;

if(argc > 1)
  {
  f = fopen(argv[1],"r");
  if(f == NULL)
    {
    fprintf(stderr,"Can't open file \"%s\"\n",argv[1]);
    exit(1);
    }
  }
else
  {
  fprintf(stderr,"No file specified\n");
  exit(1);
  }
}
```

NAME

exp	Exponential function
log	Natural logarithm function
log10	Base 10 logarithm function
pow	Power function
pow2	Compute 2**x
sqrt	Square root function

SYNOPSIS

```
#include <math.h>

r = exp(x);
r = log(x);
r = log10(x);
r = pow(x,y);
r = sqrt(x);
r = pow2(x);

double r, x, y;
```

DESCRIPTION

The *exp* function raises the natural logarithm base E to the **x** power, and **pow** raises **x** to the **y** power. For **pow**, the **x** value must be an integer if it is negative. If it is not integral, *matherr* is called with a DOMAIN error.

The *pow2* function computes 2**x by calling the "pow" function. The return value r is the value 2**x.

The *log* and *log10* functions take the base E and base 10 logarithm, respectively. Each of these as well as *sqrt*, requires a positive argument. If a negative argument is supplied, *matherr* will be called with a DOMAIN error.

SEE

matherr

Class: ANSI

NAME

fclose	Close a Level 2 file
fcloseall	Close all Level 2 files

SYNOPSIS

```
#include <stdio.h>

ret = fclose(fp);
num = fcloseall();

int ret;      return code
int num;      number of files closed
FILE *fp;     file pointer for file to be closed
```

DESCRIPTION

The *fopen* function completes the processing of a Level 2 file and releases all related resources. If the file is in the course of being written, any data which has accumulated in the buffer is written to the file, and the level 1 *close* function is called for the associated file descriptor. The buffer associated with the file block is freed.

Even though *fclose* is automatically called for all open files when your program terminates or calls *exit*, it is considered professional programming practice to close your own files explicity. The last buffer is not written until *fclose* is called and data may be lost if an output file is not properly closed.

The *fcloseall* function closes all Level 2 files that were open and returns that number. However, if an error occurs on any file, *fcloseall* continues to close the other files and then returns a value of -1.

RETURNS

Both functions return -1 to indicate an error. For success, *fclose* returns 0, and *fcloseall* returns the number of files that were closed. If -1 is returned, additional error information can be found in *errno* and *_OSERR*. If -1 is returned, additional error information can be found in *errno* and *_OSERR*.

NOTE: *fcloseall* closes the standard files **stdin**, **stdout** and **stderr**, This means, that functions such as *printf* and *perror* will fail after you call *fcloseall*.

SEE

fopen, errno, _OSERR

NAME

fdopen Attach Level 1 file to Level 2

SYNOPSIS

```
#include <stdio.h>

fp = fdopen(fh,mode);

FILE *fp;      file pointer
int fh;        file handle
char *mode;    access mode
```

DESCRIPTION

This function attaches a level 1 file to level 2. In other words, if you have used *open* to prepare a file for level 1 I/O processing, you can subsequently use level 2 I/O with that file via *fdopen*. The file handle is the value returned by the *open* function, and the access mode has the same form as described for *fopen*.

RETURNS

If the operation is successful, the function returns a non-null file pointer. Alternatively, it returns a null pointer and places error information in *errno* and *_OSERR*.

SEE

fopen, errno, _OSERR

NAME

feof	Check for Level 2 end-of-file
ferror	Check for Level 2 error

SYNOPSIS

```
#include <stdio.h>

ret = feof(fp);
ret = ferror(fp);

int ret;      non-zero if condition is true
FILE *fp;     file pointer
```

DESCRIPTION

These functions generate a non-zero value if the indicated condition is true for the specified file.

RETURNS

The return value is 0 if the condition is false, that is, no error for *ferror* or no end-of-file for *feof*. If the condition is true, a non-zero value is returned.

> NOTE: These functions are implemented as macros. Also, they do not check that **fp** is a valid file pointer.

NAME

fflush	Flush a Level 2 output buffer
flushall	Flush all Level 2 output buffers

SYNOPSIS

```
#include <stdio.h>

ret = fflush(fp);
num = flushall();

FILE *fp;      file pointer
int ret;       return code
int num;       number of open files
```

DESCRIPTION

The *fflush* macro flushes the output buffer of the specified Level 2 file. That is, it writes the buffer if the file is opened for output and the buffer contains any pending data. If an error occurs, the return value is EOF and the appropriate error code is placed into *errno*.

The *flushall* function flushes all Level 2 output buffers and returns the number of Level 2 files that are open. If an error occurs, the function continues to flush the remaining files and then returns a value of -1.

RETURNS

As noted above. In the event of a -1 return, error information can be found in *errno* and *_OSERR*.

SEE

fopen, fclose, errno, _OSERR

NAME

fgetc	Get a character from a file
getc	Get a character from a file
getchar	Get a character from stdin
fgetchar	Get a character from stdin

SYNOPSIS

```
#include <stdio.h>

c = fgetc(fp);
c = getc(fp);
c = getchar();
c = fgetchar();

int c;            return character or code
FILE *fp;         file pointer
```

DESCRIPTION

These functions get a single character from the specified Level 2 file (**stdin**) for *fgetchar* and *getchar*). Note that *getc* and *getchar* are actually implemented as macros in order to maximize execution speed.

RETURNS

Upon success, the next input character is returned. Otherwise, the functions return EOF, which is defined in stdio.h. In the event of an EOF return, error information can be found in *errno* and *_OSERR*. Most programmers treat any EOF return as an indication of end-of-file. However, if you want to distinguish errors from an end-of-file condition, you should reset *errno* before calling the function and then analyze its contents when you receive an EOF return.

SEE

fopen, errno, _OSERR

NAME

fgets	Get string from Level 2 file
gets	Get string from stdin

SYNOPSIS

```
#include <stdio.h>

p = fgets(buffer,length,fp);
p = gets(buffer);

char *p;           buffer pointer or NULL
char *buffer;      buffer pointer
int length;        buffer length in bytes
FILE *fp;          file pointer
```

DESCRIPTION

The *fgets* function gets a string from the specified Level 2 file, which must have been previously opened for input. Characters are copied from the file to the buffer until a newline ('\n') has been copied, or the buffer is full, or the end-of-file is hit. In the newline case, a null byte ('\0') is placed into the buffer after the newline if the buffer has room. In the end-of-file case, a null byte is placed into the buffer after the last byte that was read. If the end-of-file is hit before any bytes are read, a NULL pointer is returned.

The *gets* function copies characters from **stdin** (the standard input file) until a newline is reached. The newline is not copied to the buffer, but a null byte ('\0') is put there in its place.

> NOTE: *fgets* will not return a null-terminated string if **length** characters have already been placed into the buffer. Also, make sure that your *gets* buffer can hold the largest line that will be encountered while reading **stdin**, because the function does not have any way to check for a maximum length.

RETURNS

Both functions return the **buffer** argument unless an end-of-file or I/O error occurs, in which case a NULL pointer is returned.

SEE

errno, fopen, feof, ferror, fgetc, getc

EXAMPLE

```
/*
 *
 * Assume that stdin contains the following lines:
 *
 *  Hello, folks!
 *  Goodbye, folks!
 *
 */
#include <stdio.h>

main()
{
char *p,b[80];

/* For the next two lines, p will point to b */

p = gets(b);

/*  now b contains "Hello, folks!" */

p = fgets(b,sizeof(b),stdio);

/*  now b contains "Goodbye, folks!\n" */

/* After the next line, p is NULL */

p = gets(b);
}
```

NAME

fileno Get file number for a Level 2 file

SYNOPSIS

```
#include <stdio.h>

fh = fileno(fp);

int fh;         file handle
FILE *fp;       file pointer
```

DESCRIPTION

This function returns the file handle (i.e. the file number) associated with the specified file pointer. The file pointer must be one that was returned by *fopen*, *freopen* or *fdopen*.

RETURNS

As noted above.

This function is implemented as a macro. Also, it does not check that **fp** is a valid file pointer.

NAME

findpath Locate file in the current path

SYNOPSIS

```
#include <dos.h>
lock = findpath(filename)

BPTR   lock;
char   *filename;
```

DESCRIPTION

This function locates a file in the currently defined path. If the file is found, a lock (which must be unlocked) on that directory is returned (even if it is the current directory). If the file cannot be located, a -1 is returned (NULL is not used, since that is a valid lock). If the process is not a CLI process, it uses the path in effect when Workbench was loaded.

SEE

getpath, UnLock (AmigaDOS Technical Reference)

NAME

fmod	Compute floating point modulus
modf	Split floating point value

SYNOPSIS

```
#include <math.h>

x = fmod(y,z);
x = modf(y,p);

double x,y,z,*p;
```

DESCRIPTION

The *fmod* function returns **y** if **z** is 0. Otherwise, it returns a value that has the same sign as **y**, is less than **z**, and satisfies the relationship:

```
y = (i * z) + x
```

where **i** is an integer. If the % operation were defined for floating point numbers, the expression would produce:

```
x = y % z;
```

The *modf* function separates the integral and fractional parts of **y** and returns them as two doubles. The function return value is the fractional part, and the integral part is placed in the double pointed to by **p**. Both parts have the same sign as **y**. Note that the fractional part is the number that would be obtained by calling *fmod* in the following way:

```
x = fmod(y,1.0);
```

NOTE: Ensure that the second argument of *modf* is a pointer to a double. A common error is to use a pointer to an integer.

EXAMPLE

```
#include <math.h>

double r,ff,fi;

r = fmod(5.7,1.5);       /* r contains 1.2 */

ff = modf(r,&fi);        /* ff contains 0.2 */
                         /* fi contains 1.0 */
```

NAME

fmode Change mode of Level 2 file

SYNOPSIS

```
#include <stdio.h>

fmode(fp,mode);

FILE *fp;          file pointer
int mode;          0 => mode A
                   1 => mode B
```

DESCRIPTION

This function is used to change the translation mode of a Level 2 file that has been opened via *fopen*, *freopen* or *fdopen*.

In mode A, carriage returns are deleted on input, and a carriage return is inserted before each line feed on output. Mode A also detects the Ctrl-Z character (0x1A) as a logical end of file mark. In mode B, all data is transferred with no changes.

NOTE: The file pointer is not checked for validity.

SEE

fopen, freopen, fdopen

NAME

DiskfontBase		Disk font library vector

SYNOPSIS

```
extern long DiskfontBase;
DiskfontBase = OpenLibrary("diskfont.library",ver);
```

DESCRIPTION

This external location is used by various Amiga library routines which interface with the ROM Kernel text functions which deal with new fonts. It must be initialized by an **OpenLibrary** call before any of the disk font functions documented in the Amiga ROM Kernel manuals can be called. It is expected to contain the base address of the disk font library vector table.

NAME

fopen Open a Level 2 file

SYNOPSIS

```
#include <stdio.h>

fp = fopen(name, mode);

FILE *fp;       file pointer
char *name;     file name
char *mode;     access mode
```

DESCRIPTION

This function opens a file for buffered access. The **name** string can be any valid file name and may include a device code and directory path. The **mode** string indicates how the file is to be processed, as follows:

MODE	CREATE	TRUNC	READ	WRITE	APPEND	TRANSLATE
"r"	No	No	Yes	No	No	Default
"w"	Yes	Yes	No	Yes	No	Default
"a"	Yes	No	No	No	Yes	Default
"r+"	No	No	Yes	Yes	No	Default
"w+"	Yes	No	Yes	Yes	No	Default
"a+"	Yes	No	Yes	No	Yes	Default
"ra"	No	No	Yes	No	No	Mode A
"wa"	Yes	Yes	No	Yes	No	Mode A
"aa"	Yes	No	No	No	Yes	Mode A
"ra+"	No	No	Yes	Yes	No	Mode A
"wa+"	Yes	No	Yes	Yes	No	Mode A
"aa+"	Yes	No	Yes	No	Yes	Mode A
"rb"	No	No	Yes	No	No	Mode B
"wb"	Yes	Yes	No	Yes	No	Mode B
"ab"	Yes	No	No	No	Yes	Mode B
"rb+"	No	No	Yes	Yes	No	Mode B
"wb+"	Yes	No	Yes	Yes	No	Mode B
"ab+"	Yes	No	Yes	No	Yes	Mode B

The following comments explain the columns in the previous table:

CREATE	Yes	The file will be created if it does not already exist.
CREATE	No	The function will fail if the file does not already exist.
TRUNC	Yes	If the file exists, it will be truncated (i.e. marked as empty).
TRUNC	No	If the file exists, its current contents will not be disturbed.
READ	Yes	The file can be read via functions such as *fread* and *fgetc*. Also, *fseek* can be used to position the file before reading.
READ	No	The file cannot be read.
WRITE	Yes	The file can be written via functions such as *fwrite* and *fputc*. Also, *fseek* can be used to positon the file before writing.
WRITE	No	The file cannot be written, but see APPEND below.
APPEND	Yes	The file can be written, but it is automatically positioned to the current end-of-file before each write operation. This effectively prevents existing data from being changed.
APPEND	No	Automatic positioning to the end-of-file is not done before a write operation. Also, writes are not allowed unless WRITE is *'Yes'*.

TRANSLATE	Default	The external integer *_fmode* is used to set mode A or mode B as follows:
		`if (_fmode & 0x8000) set mode B`
		`else set mode A`
		For AmigaDOS, *_fmode* is normally 0x8000.

TRANSLATE	Mode A	On a read operation, each carriage return character ('\r') is deleted, and the Ctrl-Z character is treated as a logical end-of-file mark. On a write operation, each line feed character ('\n') is expanded to a carriage return followed by a line feed.
TRANSLATE	Mode B	The data is unchanged as it is read or written.

If the file is successfully opened, the function returns a pointer to a *'buffered I/O control block'*, which is defined in the header file **stdio.h**. Normally you will not need to access any information in the control block directly, but you should be very careful not to disturb the block accidentally. A common C programming error is to accidentally mutilate one of these control blocks, which can cause garbage to be written into a file.

RETURNS

A NULL pointer is returned if the file cannot be opened. Consult *errno* and *_OSERR* for detailed error information.

> When a file is opened for both reading and writing, you should call *fseek* or *rewind* when switching from reading to writing or vice-versa. It is not necessary to do this when you begin writing after reading up to the end of the file.

SEE

fclose, fdopen, fgetc, fgets, fputc, fputs, fread, freopen, fwrite

NAME

forkl	Fork with arg list
forkv	Fork with arg vector
wait	Wait for child process to complete
waitm	Wait for multiple child processes

SYNOPSIS

```
#include <dos.h>

error = forkl(prog,arg0,arg1,...,argn,NULL,env,procid);
error = forkv(prog,argv,env,procid);

cc = wait(procid);
complist = waitm(proclist);

int error;          error code
char *prog;         program name
char *arg0;         argument #0
char *arg1;         argument #1
char *argn;         argument #n
char *argv[ ];      argument vector

struct FORKENV *env;        pointer to pseudo environment
                            structure (may be NULL)
struct ProcID *procid;      pointer to process ID structure
struct ProcID **proclist;   address of pointer to linked
                            list of process ID's
struct ProcID *complist;    pointer to linked list of
                            completed process IDs
```

DESCRIPTION

These functions create a "child process" by loading a new program as a concurrent process. When the child process completes, the current program (i.e. the "parent process") can obtain its completion code via the *wait()* or *waitm()* function. The parent and child are multiprogrammed; that is, the parent continues to execute until it calls either the *wait()* or *waitm()* function.

You can specify the arguments for the new program in two ways. For the "list method", the function call includes a list of argument string pointers terminated by a NULL pointer. For the "vector method", the function call includes a single pointer to an array of argument string pointers, with the array being terminated by a NULL pointer. Following UNIX conventions, the first argument (i.e. arg0 or argv[0]) should be the program name and is normally the same as prog. Under AmigaDOS, the arguments are all concatenated into a pseudo-command line, with a blank separating adjacent arguments and a carriage return character at the end. The maximum size of this line is 255 bytes.

The pseudo environment pointer, if specified, contains optional data pertaining to default files and process execution as follows:

```
struct FORKENV {
  long priority;   /* new process priority              */
  long stack;      /* stack size for new process        */
  long stdin;      /* stdin file handle for new process  */
  long stdout;     /* stdout file handle for new process */
  long console;    /* console window for new process     */
struct MsgPort *msgport;  /* msg port to receive
                             termination*/
        };        /* message from child                */
```

The child process will be supplied with default values for any field left NULL. In addition, a NULL pointer may be passed for this parameter, which will cause default values to be used for all items. If the default values are used, the child process is created with a priority of 0, a stack size of 8000 bytes, the current **stdin**, **stdout**, and console for the parent process, and a new message port is created to receive the termination message.

The new process executes as a CLI type task. This means it will be expecting file handles for **stdin** and **stdout** to be present, and a console task handler to exist. If the parent process is running as a Workbench process then it is possible for none of these to exist for the child process to inherit. If the parent process may be invoked from Workbench then extra effort should be made to ensure the presence of file handles the child may require.

The optional message port field is provided to allow the parent process to detect when a child process has terminated while awaiting other events, such as Intuition menu events. The termination message format is similar to an Intuition message.

```
struct TermMsg {        /* termination message from child */
 struct Message msg;
 long class;            /* class == 0                     */
  short type;           /* message type == 0              */
  struct Process *process; /* process ID of sending task */
  long ret;             /* return value                   */
      };
```

When a termination message is received the parent process must still call the *wait()* function to remove the message and unload the child process. If the message was removed from the port it must be replaced by calling **PutMsg()** before calling *wait()*.

The *forkl()* and *forkv()* functions load the program from the current path using the AmigaDOS search procedure. Under Workbench version 1.1 only the current directory and the C: directory are searched for the program. Under Workbench 1.2 and beyond, directories or devices that have been added to the path are also searched. Alternatively, a path specification can be included as part of the program name.

Upon succesful creation of the child process, the fork() and forkl() functions fill in the process ID structure. The address of this structure must be passed as the last argument. The process ID structure is defined in dos.h and has the following format:

```
struct ProcID {          /* packet returned from fork() */
  struct ProcID *nextID;  /* link to next packet        */
  struct Process *process; /* process ID of child       */
  int UserPortFlag;
  struct MsgPort *parent; /* termination msg destination */
  struct MsgPort *child;  /* child process' task msg port*/
  long seglist;           /* child process' segment list */
       };
```

The nextID field may be used to link process ID structures into a simple linked list for use with the waitm() function. The process field is the address of the process's task structure, and is the value used with ROM Kernel functions such as *Signal()* that require a task ID parameter. The remaining fields are used by the *wait()* functions.

The *wait()* or *waitm()* function must be called for each child process to ensure that the child process terminates cleanly. The *wait()* function takes as its single argument a pointer to the ProcID structure for the child task. It waits for a termination message from one particular process, replying to and discarding any other messages that arrive at the message port. The function returns the child process's completion code. This is the value that was passed to the *exit()* function when the child terminated.

If any child processes share a message port, however, then *waitm()* should be called to ensure that no termination messages are lost. The *waitm()* function requires the address of a pointer to the first ProcID structure in a linked list of child process ProcID structures. It waits for one or more termination messages, removing the ProcID structure from the original linked list and inserting it into a linked list of terminated process ProcID structures. The completion code is placed in the UserPortFlag field of the structure. The function then returns a pointer to the first structure in the list of terminated process structures. Since the original list is updated, it may be reused to wait for the remaining child processes.

> NOTE: The *wait()* or *waitm()* function MUST be called for each child process before terminating the parent process. Otherwise, the child process will never be unloaded, and there is an excellent possibility that the system will crash.

RETURNS

If the specified program file cannot be found, a -1 return is made, and additional error information can be found in errno and _OSERR. Note that you must call the *wait* function in order to obtain the completion code from the child process.

SEE

AmigaDOS functions LoadSeg and CreateProc, exit, wait

EXAMPLE

```
/**
*
* This program creates a child process and displays the
* return code.  The child program name and arguments
* are taken from the command line
*
*/
#include <dos.h>
#include <stdio.h>

struct ProcID child;

void main(argc,argv)
int argc;
char *argv[ ];
{
int ret;
if (argc < 2)
        {
        printf("no program specified\n");
        printf("usage: fork program [arg1] ... [argn]\n");
        exit(0);
        }
printf("parent: beginning fork of %s\n",argv[1]);
if (forkv(argv[1],&argv[1],NULL,&child) == -1)
        {
        printf("error forking child\n");
        exit(0);
        }
else ret = wait(&child);
printf("parent: %s finished, ret = %d\n",argv[1],ret);
}

/*
* This example forks multiple child processes
```

```
*/

#include "dos.h"
#include "stdio.h"

char *child1_argv[ ] ={"task1",        /* program name */
                       "argument1",    /* 1st argument */
                       "argument2",    /* etc., etc.   */
                       NULL};

char *child2_argv[ ] ={"task2",        /* program name */
                       "argument1",    /* 1st argument */
                       "argument2",    /* etc., etc.   */
                       NULL};

char *child3_argv[ ] ={"task3",        /* program name */
                       "argument1",    /* 1st argument */
                       "argument2",    /* etc., etc.   */
                       NULL};
void main()
{
struct ProcID *children, *terminated, *task;
struct ProcID child1, child2, child3;
int taskno;

if (forkv(child1_argv[0],child1_argv,NULL,&child1) == -1)
   {
        printf("error forking child1\n");
        exit (1);
   }

if forkv(child2_argv[0],child2_argv,NULL,&child2) == -1)
   {
        printf("error forking child2\n");
        exit (1);
   }

if forkv(child3_argv[0],child3_argv,NULL,&child3) == -1
   {
        printf("error forking child3\n");
        exit (1);
   }
```

```
child3.nextID = NULL;
child2.nextID = &child3;
child1.nextID = &child2;

children = &child1;

while(children)              /* wait until no more children
*/
    {
    terminated = waitm(&children);  /* must pass ADDRESS of
*/
                                    /* pointer */
    for (task = terminated; task != NULL;
       task = task->nextID)
        {
        if (task == &child1) taskno = 1;
        else if (task == &child2) taskno = 2;
        else if (task == &child3) taskno = 3;
        printf("task %d terminated, value = %d\n",
                taskno,task->UserPortFlag);
        }
    }
    exit (0);
    return;
}
```

NAME

fprintf	Formatted print to a file
printf	Formatted printf to stdout
sprintf	Formatted print to a string

SYNOPSIS

```
#include <stdio.h>

length = fprintf(fp,fmt,arg1,arg2,...);
length = printf(fmt,arg1,arg2,...);
length = sprintf(s,fmt,arg1,arg2,...);

int length;     number of characters generated
char *fmt;      format string
FILE *fp;       file pointer
char *s;        pointer to a character string
```

```
See below for arg1, arg2, and so on.
```

DESCRIPTION

These functions generate a stream of ASCII characters by analyzing the format string and performing various conversion operations on the remaining arguments. The *printf* form sends the output stream to the Level 2 file named **stdout**, which is usually the user's screen (i.e. the "console"). The *fprintf* form is similar to *printf*, but it sends the stream to the Level 2 file specified by **fp**. Finally, the *sprintf* form places the output characters into the storage area whose address is given by s. You must ensure that this area is large enough to hold the maximum number of characters that might be generated. Note that *sprintf* also generates a null byte to terminate the stored string.

Note that *printf* has a built-in version which is functionally equivalent to the standard library version. The effect of a call to *printf* is that the most efficient internal version of the *printf* function is used.

The **fmt** argument points to a string consisting of ordinary characters and

conversion specifications. The ordinary characters are simply copied to the output, but each conversion specification is replaced by the results of the conversion. These results come from operating sequentially upon the arguments that follow **fmt**. That is, the first conversion specification operates upon **arg1**, the second operates upon **arg2**, and so on. In some cases, as described below, a conversion specification may process more than one argument.

Each conversion specification must begin with a percent character (%). If you want to place an ordinary percent into the output stream, precede it with another percent in the **fmt** string. That is, %% will send a single percent character to the output stream. If the percent is not preceded by a percent, then it introduces a conversion specification, as follows:

```
%[flags][width][.precision][l]type
```

where the brackets [...] indicate optional fields, and the fields have the following definitions:

flags Controls output justification and the printing of signs, blanks, decimal places, and hexadecimal prefixes.

width Specifies the *field width*, which is the minimum number of characters to be generated for this format item.

precision Specifies the *field precision*, which is the required precision of numeric conversions or the maximum number of characters to be copied from a string, depending on the **type** field.

size Can be either *l* for "large size" or *h* for "small size". (The *h* comes from UNIX implementations where it means "halfword".) The letter l indicates that the argument is of "large size".

type Specifies the type of argument conversion to be done.

FLAGS...

If any flag characters are used, they must appear after the percent and can be any of the following:

Minus (-) This causes the result to be left-adjusted within the field specified by **width** or within the default width.

Plus (+) This flag is used in conjunction with the various numeric conversion types to cause a plus or minus sign to be placed before the result. If it is absent, the sign character is generated only for a negative number.

Blank This flag is similar to the plus, but it causes a leading blank for a positive number and a minus sign for a negative number. If both the plus and the blank flags are present, the plus takes precedence.

Sharp (#) This flag causes special formatting. With the 'o', 'x' and 'X' types, the sharp flag prefixes any non-zero output with 0, 0x, or 0X, respectively. With the 'f', 'e' and 'E' types, the sharp flag forces the result to contain a decimal point. With the g and G types, the sharp flag forces the result to contain a decimal point and also prevents the elimination of trailing zeroes.

WIDTH...

The **width** is a non-negative number that specifies the minimum field width. If fewer characters are generated by the conversion operation, the result is padded on the left or right (depending on the minus flag described above). A blank is used as the padding character unless **width** begins with a zero. In that case, zero-padding is performed. Note that **width** specifies the minimum field width, and it will not cause lengthy output to be truncated. Use the **precision** specifier for that purpose.

If you don't want to specify the field width as a constant in the format string, you can code it as an asterisk (*), with or without a leading zero. The asterisk indicates that the width value is an integer in the argument list. See the examples for more information on this technique.

PRECISION...

The meaning of the **precision** item depends on the field type, as follows:

Type c
The precision item is ignored.

Types d, o, u, x and X
The precision is the minimum number of digits to appear. If fewer digits are generated, leading zeroes are supplied.

Types e, E and f
The precision is the number of digits to appear after the decimal point. If fewer digits are generated, trailing zeroes are supplied.

Types g and G
The precision is the maximum number of significant digits.

Type s
The precision is the maximum number of characters to be copied from the string.

As with the width item, you can use an asterisk for the precision to indicate that the value should be picked up from the next argument.

CONVERSION TYPE...

The conversion type can be any of the following:

c => Single character conversion
The associated argument must be an integer. The single character in the rightmost byte of the integer is copied to the output.

d => Decimal integer conversion
The associated argument must be an integer, and the result is a string of digit characters preceded by a sign. If the plus and blank flags are absent, the sign is produced only for a negative integer. If the *large size* modifier is present, the argument is taken as a long integer.

e = > Double conversion -d.dde-ddd
The associated argument must be a double, and the result has the form

```
-d.dde-ddd
```

where **d** is a single decimal digit, **dd** is one or more digits, and **ddd** is an exponent of exactly three digits. The first minus sign is omitted if the floating point number is positive, and the second minus sign is omitted if the exponent is positive. The plus and blank flags dictate whether there will be a sign character emitted if the number is positive. The *large size* modifier is ignored.

E = > Double conversion -d.ddE-ddd
This is exactly the same as type **e** except that the result has the form

```
-d.ddE-ddd
```

f = > Double conversion -dd.dd
The associated argument must be a double, and the result has the form

```
-dd.dd
```

where **dd** indicates one or more decimal digits. The minus sign is omitted if the number is positive, but a sign character will still be generated if the plus or blank flag is present. The number of digits before the decimal point depends on the magnitude of the number, and the number after the decimal point depends on the requested precision. If no precision is specified, the default is six decimal places. If the precision is specified as 0, or if there are no non-zero digits to the right of the decimal point, then the decimal point is omitted.

g = > Double conversion, general form
The associated argument must be a double, and the result is in the **e** or **f** format, depending on which gives the most compact result. The **e** format is used only when the exponent is less than -4 or greater than the specified or default precision. Trailing zeroes are eliminated, and the decimal point appears only if any non-zero digits follow it.

G = > Double conversion, general form
This is identical to the **g** format, except that the **E** type is used instead of **e**.

o = > Octal integer conversion
The associated argument is taken as an unsigned integer, and it is converted to a string of octal digits. If the *large size* modifier is present, the argument must be a long integer.

p = > Pointer conversion
The associated argument is taken as a data pointer, and it is converted to hexadecimal representation. Under AmigaDOS, the pointer is printed as 8 hexadecimal digits, with leading zeroes if necessary.

P = > Pointer conversion
This is the same as the **p** format, except that upper case letters are used as hexadecimal digits.

s = > String conversion
The associated argument must point to a null-terminated character string. The string is copied to the output, but the null byte is not copied.

u = > Unsigned decimal integer conversion
The associated argument is taken as an unsigned integer, and it is converted to a string of decimal digits. If the *large size* modifier is present, the argument must be a long integer.

x = > Hexadecimal integer conversion
The associated argument is take as an unsigned integer, and it is converted to a string of hexadecimal digits with lower case letters. If the *large size* modifier is present, the argument is taken as a long integer.

X = > Hexadecimal integer conversion
This is the same as the **x** format, except that upper case letters are used as hexadecimal digits.

RETURNS

Each function returns the number of output characters generated. For *sprintf*, this number does not include the terminating null byte.

EXAMPLE

```
/*
*
* This example prints a message indicated whether
* the function argument is positive or negative.
* In the second "printf", the width and precision
* are 15 and 8, respectively.
*
*/
#include <stdio.h>

pneg(value)
double value;
{
char *sign;

if(value < 0) sign = "negative";
else sign = "not negative";

printf("The number %E is %s.\n",value,sign);

printf("The number %*.*E is %s.\n",15,8,value,sign);
}
```

NAME

fputc	Put a character to a level 2 file
putc	Put a character to a level 2 file
putchar	Put a character to stdout
fputchar	Put a character to stdout

SYNOPSIS

```
#include <stdio.h>

r = fputc(c,fp);
r = putc(c,fp);
r = putchar(c);
r = fputchar(c);

int r;          EOF or c
int c;          Character to be output
FILE *fp;       Level 2 file pointer
```

DESCRIPTION

These functions put a single character to the specified Level 2 file (**stdout** for *fputchar* and *putchar*). Note that *putc* and *putchar* are actually implemented as macros in order to maximize execution speed.

RETURNS

r = c if successful
r = EOF if error

For disk files, an EOF return usually means that the disk is full. However, this type of return can also occur if the device is write-protected or if a write error occurs. In any case, additional error information can be found in *errno* and *_OSERR*.

SEE

fopen, errno, _OSERR

NAME

fputs	Put string to Level 2 file
puts	Put string to stdout

SYNOPSIS

```
#include <stdio.h>

error = fputs(s,fp);
error = puts(s);

int error;      non-zero if error
char *s;        string pointer
FILE *fp;       file pointer
```

DESCRIPTION

The *fputs* function copies string **s** to a level 2 file that was previously opened for output. The string must be terminated by a null byte, which is not copied.

The *puts* function copies string **s** to **stdout** (the standard output file). The terminating null byte is not copied, but a newline is sent after the string.

RETURNS

If an error occurs, the return value is -1; otherwise, it is 0. Additional error information can be found in *errno* and *_OSERR*.

SEE

errno, ferror, fopen, fputc

EXAMPLE

```
/*
*
* This examples writes the following two lines to stdout:
*
* This is the first line
```

```
* This is the second line
*
*/
#include <stdio.h>

main()
{
puts("This is the first line");
fputs("This is ",stdout);
puts("the second line");
}
```

NAME

fread	Read blocks from a Level 2 file
fwrite	Write blocks to a Level 2 file

SYNOPSIS

```
#include <stdio.h>

a = fread(b,bsize,n,fp);
a = fwrite(b,bsize,n,fp);

int a;          actual number of blocks
char *b;        pointer to first block
int bsize;      size of block in bytes
int n;          maximum number of blocks
FILE *fp;       file pointer
```

DESCRIPTION

These functions perform Level 2 I/O operations to read and write blocks of data. Each block contains **bsize** bytes and up to **n** blocks are stored into contiguous memory locations beginning at location **b**.

For *fread*, blocks are read until **n** have been stored or until the end-of-file is hit. If the end-of-file is hit in the middle of a block, that partial block will be stored in the **b** array, but it will not be included in the function return value. In other words, the return value indicates the number of complete blocks that were read.

For *fwrite*, blocks are written until **n** have been sent or until the output device cannot accept any more. If the output device becomes full in the middle of a block, a partial block will be written, but it will not be included in the function return value. In other words, the return value indicates the number of complete blocks that were written.

RETURNS

The functions return the number of complete blocks that were processed.

SEE

fopen, fclose, ferror, feof, fgetc, fputc, fseek

NAME

freopen Reopen a Level 2 file

SYNOPSIS

```
#include <stdio.h>

fpr = freopen(name, mode, fp);
FILE *fpr;          file pointer after re-opening
char *name;         file name
char *mode;         access mode
FILE *fp;           current file pointer
```

DESCRIPTION

This function reopens a Level 2 file. That is, it attaches a new file to a previously used file pointer. The previous file is automatically closed before the file pointer is reused. The name and mode arguments are the same as those for *fopen*.

RETURNS

fpr = NULL if error
fpr = fp if successful

> NOTE: The return code should be checked for NULL; the same errors as defined for *fopen* may occur. Also, for complete portability, do not assume that **fpr** and **fp** are identical. Use **fpr** to access the reopened file, not **fp**.

SEE

fopen, fdopen

NAME

frexp	Split fraction and exponent
ldexp	Load exponent

SYNOPSIS

```
#include <math.h>

f = frexp(v,xp);
v = ldexp(m,x);

double f;       fraction
double v;       value
int *xp;        exponent pointer
int x;          exponent
```

DESCRIPTION

The *frexp* function splits floating point value **v** into its fraction (mantissa) and exponent parts. The mantissa is returned as a double whose absolute value is greater than or equal to 0.5 and less than 1.0. The exponent is returned as an integer whose absolute value is less than 1024.

The *ldexp* function adds the integer **x** to the exponent in **f**, which is the same as computing

```
v = f * (2 ** x)
```

Note that if **f** and **x** are the results of *frexp*, then *ldexp* performs the reverse operation. Also, if the absolute value of the resulting exponent is greater than 1023, then *matherr* will be called with an overflow or underflow error indication.

SEE

fmod, matherr, modf

NAME

fscanf	Formatted input from a file
scanf	Formatted input from stdin
sscanf	Formatted input from a string

SYNOPSIS

```
#include <stdio.h>

n = fscanf(fp,fmt,arg1,arg2,...);
n = scanf(fmt,arg1,arg2,...);
n = sscanf(ss,fmt,arg1,arg2,...);

int n;              number of input items matched, or EOF
FILE *fp;           file pointer (fscanf only)
char *ss;           input string (sscanf only)
char *fmt;          format string
---- *argx;         pointers to input data areas
```

DESCRIPTION

These functions perform formatted input conversions on text obtained from the standard input file, a specified Level 2 file, or a string. The input characters are read and checked against the format string, which may contain any of the following:

White space
Any number of spaces, horizontal tabs, or newline characters will cause input to be read up to the next character that is not white space.

Ordinary characters
Any character that is not white space and is not the percent sign (%) must match the next input character. Use a double percent (%%) in the format string to match a single percent in the input. If there is not an exact match, scanning stops, and the function returns.

Conversion specification

This is multi-character sequence that indicates how the next input characters are to be converted. The form is:

```
%*n(l|h)t
```

where the various fields are defined as follows:

- **%** A percent sign introduces a conversion specifier. If you want to match a percent sign in the input, indicate this by a double percent (%%) in the format string.
- ***** The asterisk is optional. If present, it means that the conversion should be performed, but the result should not be stored. There should be no value pointer in the argument list for a suppressed conversion.
- **n** This is an optional decimal number that specifies the maximum input field width. This is used only with the **s** format.
- **l** The letter l is optional. If present, it indicates that a long conversion should be performed. Note that the letters l and h are alternatives.
- **h** The letter h is optional. If present, it indicates that a short conversions should be performed. If neither l nor h is specified, the default is an int.
- **t** The **t** stands for one of the following format characters: c, d, e, f, g, i, n, o, s, u and x. These are described below.

If the conversion is successful and assignment is not suppressed, the result is placed into the corresponding argument. The argument list must contain a pointer to an appropriate data item for each conversion specification that does not suppress assignment.

The function returns the number of conversion values that were assigned. This can be less than the number expected if the input characters do not agree with the format string. If an end-of-input is reached before any values are assigned, the return value is EOF.

The format characters listed above specify how the input characters are to be converted. Leading white space is skipped in all cases except the **c** conversion.

c => character
The corresponding argument must point to a character. The next input character is moved to that destination. No white space is skipped.

d => decimal number
The corresponding argument must point to an integer or to a long integer. The latter applies if the **d** is preceded by an **l**. The input characters should be decimal digits, optionally preceded by a plus or minus sign.

e,f,g => floating point
These three types are identical. The corresponding argument must point to a float or a double. The latter applies if the type letter is preceded by an **l**. The input characters must consist of the following sequence:

1. Optional leading white space.
2. An optional plus (+) or minus (-).
3. A sequence of decimal digits.
4. An optional decimal point followed by 0 or more decimal digits.
5. An optional exponent, consisting of the letter **e** or **E** followed by an optional plus or minus sign followed by 1 or more decimal digits.

This general form is shown below, where [...] indicates an optional part:

```
[whitespace][sign]digits[.digits][exponent]
```

n => character count
No input characters are read. The corresponding argument must point to an integer into which is written the number of input characters read so far.

o => octal number
An octal number is expected, and the corresponding argument should point to an integer, or to a long integer if the **o** is preceded by an **l**.

s = > string
A character string is expected, and the corresponding argument should point to a character array large enough to hold the string and a terminating null byte. The input string is terminated by white space or the end-of-input. Also, if a maximum field width is specified, the output array size should be at least that width plus 1, because the reading of input characters will stop at the field width even if no white space has been hit.

u = > unsigned number
An unsigned decimal number is expected, and the corresponding argument should point to an unsigned integer, or to an unsigned long integer if the **u** is preceded by an **l**.

x = > hexadecimal number
A hexadecimal number is expected, and the corresponding argument should point to an integer, or to a long integer if the **x** is preceded by an **l**. The hexadecimal number can begin with the characters *0x* or *0X*, and case is not significant for the hexadecimal letters.

RETURNS

The function returns the number of assignments that were made. For example, a return value of 3 indicates that conversion results were assigned to **arg1**, **arg2** and **arg3**.

NOTE: All of the result arguments (i.e. arg1, arg2, and so on) must be pointers. Also, you should not supply a pointer for any conversion specification that uses the * to suppress assignment.

NAME

fseek	Set Level 2 file position
ftell	Get Level 2 file position
rewind	Seek to beginning of Level 2 file

SYNOPSIS

```
#include <stdio.h>

error = fseek(fp,rpos,mode);
apos = ftell(fp);
error = rewind(fp);

int error;        non-zero if error
FILE *fp;         file pointer
long rpos;        relative file position
int mode;         seek mode
long apos;        absolute file position
```

DESCRIPTION

The *fseek* function moves the byte cursor of a Level 2 file to a new position. The *mode* argument must be one of the following:

0 The **rpos** argument is the number of bytes from the beginning of the file. This value must be positive.

1 The **rpos** argument is the number of bytes relative to the current position. This value can be positive or negative.

2 The **rpos** argument is the number of bytes relative to the end of the file. This value must be negative or zero.

The **rewind** function resets the specified file to its first byte and is equivalent to the following *fseek* call:

```
error = fseek(fp,0L,0);
```

rewind is implemented as a macro which calls *fseek*.

The *ftell* function returns a long value that is the current byte position in the file, relative to the beginning. It is equivalent to the following call:

```
apos = lseek(fp->_file,0L,1);
```

and it is implemented as a true function, not as a macro.

RETURNS

For *fseek*, a value of -1 is returned if an error occurs, and for *ftell* an error is indicated by a return value of -1L. In either case, *errno* and *_OSERR* contain additional error information.

SEE

fopen, errno, _OSERR, lseek, tell

NAME

gcvt Convert float to string

SYNOPSIS

```
#include <math.h>

p = gcvt(v,dig,buffer);

char *p;         points to buffer
double v;        floating point value
int dig;         number of significant digits
char *buffer;    output buffer
```

DESCRIPTION

This function converts the specified floating point value into a null-terminated string in the output buffer. The string will be in either of two formats. First *gcvt* attempts to produce **dig** significant digits in the FORTRAN F format. If that fails, it produces **dig** significant digits in the FORTRAN E format. Trailing zeroes will be eliminated if necessary.

RETURNS

The function returns a pointer to the buffer.

> NOTE: Ensure that the specified buffer is large enough when using this function.

SEE

ecvt, fcvt

EXAMPLE

```
/*
*
* This example displays 314150.0
*
*/
#include <stdlib.h>

main()
{
char s[100];

printf("%s\n",gcvt(-3.1415e5,7,s));
}
```

NAME

geta4 Establish addressability to the global data area

SYNOPSIS

```
#include <dos.h>

geta4();
```

DESCRIPTION

The *geta4* function establishes addressability to the global data area. It is identical in functionality to compiling the subroutine with the -y option or putting the __saveds keyword on the declaration. It is provided only for compatibility with other compilers. The -y option and __saveds keyword are preferred over *geta4*.

RETURNS

None.

NAME

getasn Get assigned device

SYNOPSIS

```
#include <stdlib.h>

var = getasn(name);

char *var;      environment variable pointer or NULL
char *name;     environment variable name
```

DESCRIPTION

This function searches the environment strings for one that has the form

```
name=var
```

where *name* is the function argument. If such a string exists, the function returns a pointer to the *var* portion, which is null-terminated. Otherwise, a NULL pointer is returned.

In practice, the *getasn* function can be used to search the device list for a name. If the name is found, it returns a pointer to the device structure. This is useful for checking to see if a logical name is assigned. The following example

```
    if (getasn("INCLUDE")) open ("INCLUDE:name")
```

shows sample syntax for this function.

RETURNS

NULL is returned if *name* cannot be located in the current environment. Note that NULL is defined in *stdio.h*.

SEE

getenv, putenv

EXAMPLE

```
#include <stdlib.h>
#include <stdio.h>

char *path;

path = getasn("PATH"); /* get PATH variable */
if(path == NULL) fprintf(stderr,"No PATH variable\n");
```

NAME

getcd Get current directory

SYNOPSIS

```
#include <dos.h>

error = getcd(drive,path);

int error;        0 if successful
int drive;        drive code
char *path;       points to path area (255 bytes recommended)
```

DESCRIPTION

This function gets the current directory path for the specified disk drive. The drive specification is retained in this implementation for compatibility with other implementations. However, only a value of 0 (indicating the current drive) is supported under AmigaDOS. Any other value is considered an error.

Note that the path area must be large enough to contain the expected path. The returned string will contain the entire path, including the volume name of the device.

RETURNS

If the operation is successful, the function returns 0. Otherwise it returns -1 and places error information in *errno* and *_OSERR*.

SEE

getcwd, errno, _OSERR

NAME

getcd Get current directory
getcd Get current directory

SYNOPSIS

```
#include <dos.h>

getclk(clock);
error = chgclk(clock);

int error;
unsigned char *clock;
```

DESCRIPTION

The *getclk* function obtains the current setting of the system clock and places it into an 8-byte array as follows:

Byte		Contents
clock[0]	=>	day of week (0 for Sunday)
clock[1]	=>	year - 1980
clock[2]	=>	month (1 to 12)
clock[3]	=>	day (1 to 31)
clock[4]	=>	hour (0 to 23)
clock[5]	=>	minute (0 to 59)
clock[6]	=>	second (0 to 59)
clock[7]	=>	hundredths (0 to 99)

The *chgclk* function changes the setting of the system clock to the value specified in the array.

RETURNS

If the array is invalid or the operation failed, *chgclk* returns a non-zero value.

> NOTE: If your machine is equipped with a hardware clock, its state is not necessarily changed by a call to *chgclk*.

SEE

errno, _OSERR

NAME

getcwd — Get current working directory

SYNOPSIS

```
#include <stdio.h>

p = getcwd(b,size);

char *p;      Same as b is successful, else NULL
char *b;      Points to path buffer
int size;     Size of path buffer
```

DESCRIPTION

This function obtains the path name for the current working directory. If the buffer pointer **b** is not null, then the path string is placed there if it will fit, and the return pointer **p** is the same as **b**. If **b** is null, then *malloc* is used to obtain a buffer of **size** bytes to hold the path string. In this latter case, you should use the **free** function to release the buffer when you are finished with it.

Note that the *getcd* function is often more efficient under AmigaDOS since it does not use memory allocation.

RETURNS

If the operation is successful, the function returns a pointer to the buffer. Otherwise it returns a null pointer and places error information in *errno* and *_OSERR*. Also, a null pointer is returned if the path string will not fit in the buffer or if a buffer cannot be allocated. In either of those cases, *errno* is unchanged, and *_OSERR* is reset.

SEE

getcd, errno, _OSERR

NAME

getdfs Get free disk space

SYNOPSIS

```
#include <dos.h>

error = getdfs(drive,info);

int error;                  0 if successful
char *drive;                drive or volume name
                            (NULL => current drive)
struct DISKINFO *info;      disk information
```

DESCRIPTION

This function obtains information about the specified disk drive, including the amount of free space available. If a null pointer is passed as the drive name, information is obtained about the current drive. The DISKINFO structure is defined in **dos.h**. For AmigaDOS, this is the same as the AmigaDOS InfoData structure:

```
struct InfoData {
   long   id_NumSoftErrors;
   long   id_UnitNumber;
   long   id_DiskState;
   long   id_NumBlocks;
   long   id_NumBlocksUsed;
   long   id_BytesPerBlock;
   long   id_DiskType;
   BPTR   id_VolumeNode;
   long   id_InUse;
};
```

RETURNS

A return value of 0 indicates success. If the drive code is invalid or no disk is mounted on that drive, then the return value is -1. Note that no additional information is provided in *errno* or *_OSERR*.

EXAMPLE

```
/* Compute number of bytes available on current drive: */

#include <dos.h>
struct DISKINFO info;
long size;

if(getdfs(0,&info) == 0)
    size = (info.id_NumBlocks - info.id_NumBlocksUsed)
             * info.id_BytesPerBlock;
```

NAME

getenv Get environment variable

SYNOPSIS

```
#include <stdlib.h>

var = getenv(name);

char *var;     environment variable pointer or NULL
char *name;    environment variable name
```

DESCRIPTION

This function searches the environment strings for one that has the form

```
name=var
```

where *name* is the function argument. If such a string exists, the function returns a pointer to the *var* portion, which is null-terminated. Otherwise, a NULL pointer is returned.

Under AmigaDOS, environment variables are obtained by reading the first line out of the file *env:variable.*

The memory returned by *getenv* is obtained via *malloc* so it remains through the duration of your program. However, the next call to *getenv* will destroy the previous results from *getenv*. If you wish to return the memory, you may call *free*, but it is not required.

RETURNS

NULL is returned if *name* cannot be located in the current environment. Note that NULL is defined in *stdio.h.*

SEE

putenv

EXAMPLE

```
#include <stdlib.h>
#include <stdio.h>

char *path;

main()
{
path = getenv("PATH"); /* get PATH variable */
if(path == NULL) fprintf(stderr,"No PATH variable\n");
}
```

NAME

getfa Get file attribute

SYNOPSIS

```
#include <dos.h>

fa = getfa(name);

int fa;        1, -1, or 0
char *name;    file name
```

DESCRIPTION

This function determines whether or not the specified file is a directory. The status is returned in **fa** and contains the following information:

1	=>	Directory file
0	=>	Error
-1	=>	Normal file

RETURNS

If the operation is unsuccessful, the function returns 0 and places error information in *errno* and *_OSERR*.

SEE

errno, _OSERR

NAME

getfnl Get filename list

SYNOPSIS

```
#include <stdlib.h>

n = getfnl(fnp,fna,fnasize,attr);

int n;                  number of matching file names
char *fnp;              file name pattern
char *fna;              file name array
unsigned fnasize;       size of file name array
int attr;               file attribute
```

DESCRIPTION

This function gets all file names that match the specified pattern and attribute, and it places them into the file name array. Each name is stored as a null-terminated string, and the file name array is terminated by a null string (i.e. a string consisting of only a null byte). If the file name pattern includes a path prefix, that prefix is placed in front of each matching file name.

The function return value is the number of strings stored in the array, not including the terminating null string.

The file name pattern has the general form

```
drive:path/node
```

The function first strips off the drive and directory path portion and restricts its search to that area of the file system. The node and extension parts can contain any valid file name characters, including the wildcard pattern matching characters. AmigaDOS wildcard characters are supported by default. Setting the external integer location **msflag** to a nonzero value will cause MS-DOS wildcard characters * and ? to be supported instead. Some examples are:

df0:#?.c	Finds all files on drive df0: that have *.c* as their extension. A file named *abc.c* would thus be place in the array as *df0:abc.c*. This assumes that the msflag is set to zero.
:abc/def/q.x?*	Finds all files in the directory **:abc/def** that begin with the letter q and have extensions consisting of the letter x and one other letter. For example, one such name would be *:abc/def/queen.x*. This assumes that the msflag is set to 1.
*XYZ**	Finds all files in the current directory that begin with *XYZ*. One example is *XYZ*. This assumes that the msflag is set to 1.

Notice that AmigaDOS makes no distinction between upper and lower case in any part of the file name.

The attribute is an integer defined as follows:

0 = > non directory files
1 = > all files including directories

RETURNS

A value of -1 is returned if the file name pattern is invalid or if there is not enough room in the file name array. In the first case, **_oserr** will contain further error information.

SEE

dfind, dnext, strbpl, strsrt, _oserr

EXAMPLE

```
/*
* This program constructs an array of pointers to all normal
* files in the current directory * that have an extension of
* .c.  Then the array is sorted into ASCII order.
*
*/
#include <stdlib.h>
```

```
main()
{
char names[3000],*pointers[300];
int count;

count = getfnl("#?.c",names,sizeof(names),0);

if(count > 0)
    {
    if(strbpl(pointers,300,names) != count)
        {
        fprintf(stderr,"Too many file names\n");
        exit(1);
        }
    strsrt(pointers,count);
    }
else
    {
    if(_OSERR) poserr("FILES");
    else fprintf(stderr,"Too many files\n");
    exit(1);
    }
}
```

NAME

getft Get file time

SYNOPSIS

```
#include <dos.h>

ft = getft(name);

long ft;        file time or -1 if error;
char *name;     file name
```

DESCRIPTION

This function gets the time and date information associated with the specified file. This information usually indicates when the file was created or last updated. This function returns the file time expressed as the number of seconds since 00:00:00 Greenwich Mean Time, January 1, 1970.

RETURNS

If *getft* is successful, the file time (a long integer) is returned. Otherwise a value of -1L is returned. Additional error information can be found in *errno* and *_OSERR*.

SEE

errno, _OSERR

NAME

getmem	Get Level 2 memory block (short)
getml	Get Level 2 memory block (long)

SYNOPSIS

```
#include <stdlib.h>

p = getmem(sbytes);
p = getml(lbytes);

char *p;              block pointer
unsigned sbytes;      number of bytes
long lbytes;          number of bytes
```

DESCRIPTION

These functions allocate a block from the Level 2 memory pool and return a pointer to the first byte in the block. If the pool does not currently contain a block of sufficient size, the Level 1 memory allocator, *sbrk*, is called to obtain more space from the operating system. If that step fails, a NULL pointer is returned.

Unlike the Level 3 memory allocator, Level 2 does not allocate any additional bytes to retain the block size and other overhead information. However, Level 2 does require a minimum block size of **MELTSIZE** bytes, where **MELTSIZE** is defined in the **dos.h** header file. If you request a block that is smaller, *getmem* and *getml* will round it up to the minimum size.

You will probably want to use the *malloc* function instead of *getmem*.

RETURNS

A NULL pointer is returned if the block could not be allocated. Otherwise, a character pointer is returned, but it can be cast to any other pointer type.

> NOTE: The Level 2 memory allocator does not retain any information about the blocks it has allocated, you must keep track of the block pointer and size if you plan to release the memory at some later time.

SEE

rlsmem, rlsml, sizmem

EXAMPLE

See the example for *malloc*.

NAME

getpath Get the path for a specific directory/file

SYNOPSIS

```
#include <dos.h>
error = getpath(lock, path);

int   error;     0 if ok, -1 if error
BPTR  lock;
char  *path;     destination buffer
```

DESCRIPTION

This function returns the fully specified path string for the directory/file referenced by the lock. The path includes the volume name. The destination buffer must be large enough to contain the string.

SEE

findpath, UnLock (AmigaDOS Technical Reference)

NAME

getreg	Obtain 68000-specific registers
putreg	Set up 68000-specific registers

SYNOPSIS

```
#include <dos.h>

long x = getreg(12);   obtain the current global static base
long x = putreg(12);   set up the current global static base
```

DESCRIPTION

The built-in function *getreg* takes as its parameter a constant integer in the range of 0 to 15. The number that you pass is the register number for which you want the current contents. Numbers 0 through 7 correspond to the d0-d7 registers, while numbers 8 through 15 correspond to the a0-7 registers.

The built-in function *putreg* takes as its parameter the register number as described above for *getreg*. The number that you pass is a long integer, which is placed in the specified register.

RETURNS

The *getreg* function returns the current value of the register (a long integer). The *putreg* function returns a void.

> Incorrect use of these functions can cause serious problems. These functions are intended for use with interrupt code. For instance, the *getreg* function is useful for obtaining the value of the system registers (e.g., a4) to be passed to an interrupter chain. However, the *getreg* function is <u>not</u> a reliable way of getting the value of a variable because the code generator may change code generation style during compile time. While programmers may find these functions useful in some situations, a great deal of care and skill should be exercised in their use.

NAME

GfxBase Graphics Library Vector

SYNOPSIS

```
extern long GfxBase;
GfxBase = OpenLibrary("graphics.library",ver);
```

DESCRIPTION

This external location is used by various Amiga library routines which interface with the ROM Kernel graphics system functions. It must be initialized by an **OpenLibrary** call before any of the graphics functions documented in the Amiga ROM Kernel manuals can be called. It is expected to contain the base address of the graphics library vector table.

NAME

gmtime	Unpack Greenwich Mean Time (GMT)
localtime	Unpack local time

SYNOPSIS

```
#include <time.h>

ut = gmtime(t);
ut = localtime(t);

struct tm *ut;
long *t;
```

DESCRIPTION

These functions unpack a time value from the long integer form into a structure. Normally the time value represents the number of seconds since 00:00:00, January 1, 1970, Greenwich Mean Time. The *time* function returns this kind of number. For *gmtime*, this number is converted as is, while *localtime* adjusts the number for the local time zone.

SEE

asctime, ctime, localtime, time

> NOTE: These functions expect a pointer as the argument. A common error is to pass the actual time value instead of the pointer. Also, the functions share a static data area for their return values, and a call to either one will destroy the results of the previous call.

EXAMPLE

```
#include <time.h>
main()
{
```

```
struct tm *p;
long t;

time(&t);
p = gmtime(&t);
printf("GMT is %s\n",asctime(p));
}
```

NAME

IntuitionBase Intuition Library Vector

SYNOPSIS

```
extern long IntuitionBase;
IntuitionBase = OpenLibrary("intuition.library",ver);
```

DESCRIPTION

This external location is used by various Amiga library routines that interface with the Intuition system functions. It must be initialized by an **OpenLibrary** call before any of the functions documented in the Amiga Intuition manuals can be called. It is expected to contain the base address of the Intuition library vector table.

NAME

iomode Change mode of Level 1 file

SYNOPSIS

```
#include <fcntl.h>

error = iomode(fh,mode);

int error;   error code
int fh;      file handle
int mode;    0 => translated mode
             1 => raw mode
```

DESCRIPTION

This function changes the mode of the specified Level 1 file. When in translated mode, carriage returns are deleted on input, and a carriage return is inserted before each line feed on output. In raw mode, all data in the file is transferred as is.

Note that *iomode* affects only the software translation that is done by the Level 1 I/O service functions.

RETURNS

A non-zero return value indicates that the specified file handle is not associated with a Level 1 file.

SEE

open, getfc

NAME

isalnum	Test if alphanumeric character
isalpha	Test if alphabetic character
isascii	Test if ASCII character
iscntrl	Test if control character
iscsym	Test if C symbol character
iscsymf	Test if C symbol lead character
isdigit	Test if decimal digit character
isgraph	Test if graphic character
islower	Test if lower case character
isprint	Test if printable character
ispunct	Test if punctuation character
isspace	Test if space character
isupper	Test if upper case character
isxdigit	Test if hex digit character

SYNOPSIS

```
#include <ctype.h>

t = isalnum(c);
t = isalpha(c);
t = isascii(c);
t = iscntrl(c);
t = iscsym(c);
t = iscsymf(c);
t = isdigit(c);
t = isgraph(c);
t = islower(c);
t = isprint(c);
t = ispunct(c);
t = isspace(c);
t = isupper(c);
t = isxdigit(c);

int t;   truth value (0 if false, non-zero if true)
int c:   character to test
```

DESCRIPTION

These are used to test for various character types. (It should be noted that no function version actually exists for *isascii*, *iscsym*, *iscysmf*, *isgraph*, *isprint*, *ispunct*, and *isxdigit*.)

If you include **ctype.h** as shown above, then the functions are actually defined as macros and generate in-line code to test the static array named **_ctype**. This array contains a bit mask for each of the 256 possible character values and for the integer value -1. See the **ctype.h** description for an explanation of this array.

If you don't include **ctype.h**, these functions will be resolved in the library, which can reduce your program size slightly at the expense of execution speed. If you want to use the function versions but must include **ctype.h** for some other reason, use **#undef** to undefine the appropriate character test macros.

> NOTE: You can use either characters or integers as arguments, but the macros are defined only over the integer range from -1 to 255. The functions, however, will correctly handle the entire integer range.
>
> The reason -1 is included as a valid argument is to avoid a nonsensical result if you feed the EOF value to one of the macros or functions. EOF can be returned by *getchar* and other I/O functions, and if you pass it to any of the character test functions, the resulting truth value will be zero.

SEE

_ctype

EXAMPLE

```
#include <stdio.h>
#include <ctype.h>

main()
```

```
{
char b[100];
int c;
```

```
{
while((c = getchar()) != EOF)
        printf("\n%c %s alphabetic.\n",c,
                isalpha(c) ? "is" : "is not");
}
```

NAME

longjmp	Perform long jump
setjmp	Set long jump parameters

SYNOPSIS

```
#include <setjmp.h>

ret = setjmp(save);
longjmp(save,value);

int ret;              return code
int value;            return value
jmp_buf save;        address of save area
```

DESCRIPTION

The *setjmp* function checkpoints the current stack mark in the save area and returns a code of 0. A subsequent call to *longjmp* will then cause control to return to the next statement after the original *setjmp* call, with **value** as the return code. If **value** is 0, it is forced to 1 by *longjmp*.

This mechanism is useful for quickly popping back up through multiple layers of function calls under exceptional circumstances.

RETURNS

A return code of 0 from *setjmp* indicates that this is the initial call to save the stack. A non-zero return code indicates that *longjmp* has been executed.

> NOTE: Calling *longjmp* with an invalid save area is an effective way to disrupt the system. One common error is to use *longjmp* after the function calling *setjmp* has returned to its caller. This cannot possibly succeed, since the stack frame for that function no longer exists.

NAME

lsbrk	Allocate Level 1 memory (long)
rbrk	Release Level 1 memory
sbrk	Allocate Level 1 memory (unsigned)

SYNOPSIS

```
#include <stdlib.h>

p = lsbrk(lbytes);
error = rbrk();
p = sbrk(sbytes);

char *p;            block pointer
long lbytes;        number of bytes
unsigned sbytes;    number of bytes
int error;          0 if successful
```

DESCRIPTION

These functions form Level 1 of Lattice's layered memory allocation system. This level is **incompatible** with most versions of UNIX. The memory pool is viewed as a group of non-contiguous areas of memory allocated from the system free memory pool.

The *sbrk* and *lsbrk* functions requests **sbytes** or **lbytes** of memory from the system, adding the the allocated block to a linked list of memory blocks to be returned to the system when the program terminates. The function returns the address of the block just allocated.

The *rbrk* function returns all memory from this area to the operating system.

> NOTE: Calling *rbrk* will free all memory in the memory pool, including level-2 I/O buffers.

RETURNS

If *sbrk* fails, it returns value **(char *)(-1)** which is -1 cast into a character pointer. This strange return is a legacy of UNIX. For *lsbrk*, an error is indicated by a NULL pointer.

SEE

getmem, malloc

NAME

lseek	Set Level 1 file position
tell	Get Level 1 file position

SYNOPSIS

```
#include <fcntl.h>

apos = lseek(fh,rpos,mode);
apos = tell(fh);

int fh;          file handle
long rpos;       relative file position
int mode;        seek mode
long apos;       absolute file position
```

DESCRIPTION

The *lseek* function moves the byte cursor of a Level 1 file to a new position. The **mode** argument must be one of the following:

0 The **rpos** argument is the number of bytes from the beginning of the file. This value must be positive.

1 The **rpos** argument is the number of bytes relative to the current position. This value can be positive or negative.

2 The **rpos** argument is the number of bytes relative to the end of the file. This value must be negative or zero.

If *lseek* is asked to move 0 bytes relative to the current position, it simply returns the current file position. The *tell* function is then equivalent to:

```
apos = lseek(fh,0L,1);
```

RETURNS

Both functions return -1L if an error occurs, in which case *errno* and *_OSERR* contain additional error information.

SEE

errno, _OSERR, open

EXAMPLE

```
/**
*
* This program totals the number of bytes used by
* all normal files in the current directory.
*
**/
#include <fcntl.h>   /* for Level 1 I/O */

char names[8192];    /* holds file names */

main()
{
char *p;
int f,n;
long x,y;

if(getfnl("*.*",names,sizeof(names),0) <= 0)
  {
  printf("Can't build file name list\n");
  exit(1);
  }
for(x = 0, n = 0, p = names; *p != '\0'; p += strlen(p) + 1)
  {
  f = open(p,O_RDONLY);
  if(f < 0)
    {
    printf("Can't open \"%s\"\n",p);
    exit(1);
    }
```

```
    y = lseek(f,0L,2);
    if(y < 0)
      {
      printf("Seek failure on \"%s\"\n",p);
      exit(1);
      }
    x += y;
    n++;
    close(f);
    }
  printf("%d files, %ld bytes used\n",n,x);
  }
```

NAME

main Your main or principal function

SYNOPSIS

```
#include "workbench/startup.h"

void main(argc,argv);

int argc;       argument count
union {
  char *args[ ];
  struct WBStartup *msg;
 } argv;       argument vector
```

DESCRIPTION

This function does not actually exist in the library; you must supply one of these *main programs* in each of your applications. If you trace through the two startup modules *c.a* and *_main.c*, you will find that *c.a* passes control to *_main.c*, which then calls the function named *main*. Since we supply the source code for both of these modules, you are free to change this initialization procedure for special applications. The standard version simulates UNIX's interface with C programs by setting up a *vector*, which is simply an array of pointers.

The **argv.args** array contains pointers to the command line arguments, and **argc** indicates how many pointers are in the array. For example, if you invoke **myprog** with the following command line

```
myprog abc def "ghi jkl"
```

then **argv.args** is set up as follows:

```
argv.args[0] => "myprog"
argv.args[1] => "abc"
argv.args[2] => "def"
argv.args[3] => "ghi jkl"
```

and **argc** contains the value 4.

Under WorkBench there is no command line. In this case, **argc** is 0 indicating no command or arguments, and **argv.msg** is a pointer to the WorkBench startup message structure.

RETURNS

None. When *main* returns to its caller (normally *_main.c*), the program exits to AmigaDOS with a termination code of 0. If you want to pass a non-zero termination code back to AmigaDOS, use the *exit* or *_exit* function.

SEE

exit, _exit

EXAMPLE

```
/**
*
* This program is intended to run only under CLI,
* and displays the command and any arguments
*
**/
void main(argc, argv)
int argc;
char *argv[ ];
{
int i;

printf("command = %s\n",argv[0]);
for (i = 0; argc > 1; i++, argc--)
        printf("argument %d = %s\n",i,argv[i]);
}

/**
*
* This program is intended to run only under WorkBench, and
* gets its arguments from the WorkBench message structure
**/
#include "workbench/startup.h"
```

```
void main(argc, argv)
int argc;
struct WBStartup *argv;
{
int i;

if (argc != 0) exit(ERROR);

printf("command = %s\n",argv->sm_ArgList[0]->wa_name);
for (i = 1; i < argv->sm_NumArgs; i++)
printf("argument %d = %s\n",i,argv->sm_ArgList[i]->wa_name);
}

/**
*
* This program is intended to run under either
* WorkBench or CLI.
*
**/
#include "workbench/startup.h"

void main(argc, argv)
int argc;
union  {
        char *args[ ];
        struct WBStartup *msg;
        } argv;  argument vector
{
int i;

if (argc != 0)
  {
  printf("command = %s\n",argv.args[0]);
  for (i = 0; argc > 1; i++, argc--)
      printf("argument %d = %s\n",i,argv.args[i]);
  }
else
  {
  printf("command = %s\n",
          argv.msg->sm_ArgList[0]->wa_name);
  for (i = 1; i < argv.msg->sm_NumArgs; i++)
```

```
        printf("argument %d = %s\n",i,
                argv.msg->sm_ArgList[i]->wa_name);
    }
}
```

NAME

MathBase FFP Library Vector

SYNOPSIS

```
extern long MathBase;
MathBase = OpenLibrary("mathffp.library",ver);
```

DESCRIPTION

This external location is used to interface with the Motorola Fast Floating Point library routines provided by Amiga. It is initialized by an **OpenLibrary** call when a program compiled with the FFP option performs the first floating point operation. It contains the base address of the FFP math library vector table. If you make direct calls to the Amiga FFP functions you must first initialize this location by calling **OpenLibrary**.

NAME

matherr	Math error handler
except	Call math error handler

SYNOPSIS

```
#include <math.h>

a = matherr(x);
r = except(type,name,arg1,arg2,retval);

int a;                      action code
struct exception *x;        exception vector
double r;                   actual return value
int type;                   error type
char *name;                 math function name
double arg1;                first argument
double arg2;                second argument
double retval;              proposed return value
```

DESCRIPTION

The *matherr* function is called whenever one of the higher-level math functions detects an error. The exception vector structure is defined in **math.h** and contains information about the error as follows:

```
struct exception
  {
  int type;               /* error type            */
  char *name;             /* math function name    */
  double arg1, arg2;      /* function arguments    */
  double retval;          /* proposed return value */
  };
```

The standard library version of *matherr* translates the error type into a UNIX error code that is placed into *errno*. Then the function returns an action code of 0 to indicate that the math function should simply use the proposed return value. In other words, the math function will pass that value back to its caller.

The Lattice compiler package includes source code for *matherr* so you can change it to do more sophisticated error correction. One typical change is to place a different return value into the exception vector and then return a non-zero action code. This informs the math function that the return value has been changed.

The *except* function is a Lattice extension to UNIX that simplifies the interface to *matherr* by setting up the exception vector and processing the action code and return value. It is intended to ease the error-handling chore in user-written math functions.

When your math function encounters an error, it should call *except* specifying one of the following error types, which are defined in the **math.h** header file:

Symbol	Code	Meaning
DOMAIN	1	Domain error
SING	2	Singularity
OVERFLOW	3	Overflow (number too large)
UNDERFLOW	4	Underflow (number too small)
TLOSS	5	Total loss of significance
PLOSS	6	Partial loss of significance

You can define new type codes if your application requires them, but you should then change *matherr* to perform the appropriate mapping into the UNIX error codes. The default mapping is:

matherr	errno
DOMAIN	EDOM
SING	EDOM
OVERFLOW	ERANGE
UNDERFLOW	ERANGE
TLOSS	ERANGE
PLOSS	ERANGE

RETURNS

For *matherr*, a non-zero return indicates that the proposed return value in the exception vector has been changed and that the new value should be used. A zero return indicates that the proposed return value is OK.

For *except*, the actual return value (a double) is passed back.

SEE

_fperr

NAME

max	Compute maximum of two values
min	Compute minimum of two values

SYNOPSIS

```
#include <math.h>

v = max(a,b);
v = min(a,b);
```

DESCRIPTION

These macros compute the maximum and minimum of two arithmetic values. They are defined as

```
#define max(a,b) ((a)>(b)?(a):(b))
#define min(a,b) ((a)<=(b)?(a):(b))
```

and will work with any arithmetic type or combination of types.

Note that *max* and *min* have built-in versions which are functionally equivalent to the standard library versions. The statement *#include <string.h>* provides a default setting by which built-in functions are accessed. If you don't want the built-in function, you can use an #undef statement.

NAME

MemCleanup Deallocate all allocated memory

SYNOPSIS

```
#include <stdlib.h>

MemCleanup();
```

DESCRIPTION

The *memcleanup* function traverses the linked list of allocated memory to release any memory allocated and not yet returned to the system. This is necessary because Amiga CLI does not perform any cleanup functions.

This function is normally called from the Lattice startup code as a program is terminating. You are free to replace the standard *memcleanup* function with one of your own.

RETURNS

None.

NAME

memccpy	Copy a memory block up to a char
memchr	Find a character in a memory block
memcmp	Compare two memory blocks
memcpy	Copy a memory block
memset	Set a memory block to a value
movmem	Move a memory block
repmem	Replicate values through a block
setmem	Set a memory block to a value
swmem	Swap two memory blocks

SYNOPSIS

```
#include <string.h>

s = memccpy(to,from,c,n);
s = memchr(a,c,n);
x = memcmp(a,b,n);
s = memcpy(to,from,n);
s = memset(to,c,n);

movmem(from,to,n);
repmem(to,vt,nv,nt);
setmem(to,n,c);
swmem(a,b,n);

char *to;          destination pointer
char *from;        source pointer
unsigned n;        number of bytes
char c;            character value
char *a,*b;        block pointers
char *vt;          value template
int nv;            number of bytes in template
int nt;            number of templates in block
char *s;           return pointer
int x;             return value
```

DESCRIPTION

These functions manipulate blocks of memory in various ways.

The *memcpy* and *movmem* functions are similar, except the former was introduced with UNIX V, while the latter is a traditional Lattice function. In a like manner, *memset* and *setmem* perform the same operation, except that the former is UNIX-compatible. Note that *memcpy* and *memset* return a pointer to the destination block, while *movmem* and *setmem* have void returns. Also note that *movmem* is smart enough to handle overlapping memory blocks correctly.

The *memccpy* function is similar to *memcpy* except that copying stops after the specified block size has been copied or after the specified character has been copied. It returns a pointer to the character after **c** in the **from** block, or a null pointer if **c** was not found in the first **n** characters. Note that, like *memcpy*, *memccpy* does not handle overlapping memory blocks. If you specify overlapping blocks to this function, the results are unpredictable.

The *memchr* function returns a pointer to the first occurrence of the specified character in the block, or a null pointer if the character is not found.

The *memcmp* function performs a character-by-character comparison of two memory blocks and returns an integral value as follows:

RETURN	MEANING
Negative	First block is below second
Zero	First block equals second
Positive	First block is above second

There is currently no UNIX equivalent for *swmem* and *repmem*. The former merely swaps two blocks in memory, although it has a major performance advantage over the typical for-loop approach. The latter replicates a template of values throughout a block and is very useful when you need to initialize an array of structures to some non-zero pattern.

Note that *memcmp*, *memcpy*, and *memset* have built-in versions which are functionally equivalent to the standard library versions. A built-in version generates in-line 68000 instructions without needing to make calls to the

library. The statement *#include <string.h>* provides a default setting by which any built-in functions are accessed. If you don't want a particular built-in function, you can use an #undef statement as follows: #undef memcmp.

RETURNS

As noted above.

NOTE: These functions neither recognize nor produce the null terminator byte usually found at the end of strings. A popular mistake is to assume that *memcpy*, in common with *strcpy*, automatically places a null byte at the end of the block. It does not.

Unlike previous versions of the Lattice C Compiler, memcpy is not smart enough to handle overlapping blocks. The ANSI function *memmove* should be used instead.

NAME

mkdir Make a new directory

SYNOPSIS

```
#include <stdio.h>

error = mkdir(path);

int error;      0 if successful
char *path;     points to new directory path string
```

DESCRIPTION

This function makes a new directory in the specified path. For example, if **path** is *sys:/abc/def/ghi*, then the new directory is named *ghi* and is in the path */abc/def* on the volume labeled *sys:*. For AmigaDOS, the path may begin with a drive or volume name and a colon.

RETURNS

If the operation is successful, the function returns 0. Otherwise it returns -1 and places error information in *errno* and *_OSERR*.

SEE

errno, _OSERR

NAME

msflag MS-DOS File Pattern Flag

SYNOPSIS

```
extern int msflag;
```

DESCRIPTION

This external integer is used by the file name pattern matching functions to specify AmigaDOS or MSDOS wildcard characters. If *msflag* is non-zero then MSDOS file name patterns are used. The default is to use AmigaDOS file name patterns.

SEE

dfind, getfnl

NAME

onbreak   Plant break trap

SYNOPSIS

```
#include <dos.h>

error = onbreak(func);

int error;         0 if successful
int (*func)();     pointer to function to be called
```

DESCRIPTION

This function plants a break trap, which is simply a function that gets called whenever the user keys Ctrl-C or Ctrl-D. The standard break keys Ctrl-E and Ctrl-F are ignored by *onbreak*.

The *onbreak* function can perform any AmigaDOS operations. If it returns a value of 0, then execution resumes at the interrupted point. Otherwise, the program is aborted immediately.

If **func** is null, then the current break trap, if any, is removed and the default interrupt handler is restored. With the default handler, Ctrl-C and Ctrl-D cause a requester to appear on the screen with choices of **continue** or **abort**.

Detection of Ctrl-C and Ctrl-D is performed during level 1 I/O. Explicit checks for these events can be forced by calls to the function *chkabort*.

RETURNS

The *onbreak* function returns 0 if it was successful. The break trap function should return 0 to continue execution and non-zero to abort.

SEE

chkabort, signal

EXAMPLE

This program tests the onbreak function. After the initial message is printed, you should get the "Break received" message when you hit Ctrl-C or Ctrl-D. The second time will cause the program to terminate.

```
#include <dos.h>
#include <stdio.h>
int i = 0;

int brk()                /* This is the break function */
{
printf("Break received...\n");
return(i++);
}

main()                   /* This is the main program  */
{
printf("Setting break trap...\n");
if(onbreak(&brk)) printf("Can't set break trap\n");
while(1) chkabort();     /* check for CTRL-C          */
}
```

NAME

onexit   Set an exit trap

SYNOPSIS

```
#include <stdlib.h>

success = onexit(func);

int success;         non-zero if successful
int (*func)();       pointer to trap function
```

DESCRIPTION

This function establishes a *trap* which will be called when the program terminates. The trap function is called just before the program returns to the operating system. For normal termination via the *exit* function or via a return from the *main* function, all buffers are flushed and files are closed before the trap is called. If the program is using *_exit*, the files and buffers may still be open, depending on what the program does before terminating. In both cases, user-allocated memory is not yet freed.

The *onexit* function is equivalent to the ANSI function *atexit*. As an extension to the ANSI definition of the *atexit* function, the Lattice implementation passes the exit code to the trap function as its only argument. Then whatever value the trap function returns is used as the real exit code.

Another difference between our implementation and the ANSI definition is that we allow only one trap. Each call to *onexit* overrides the previous trap. If you call *onexit* with a null pointer, the current trap is removed.

> NOTE: The exit trap is called after all files have been closed. A common mistake is to issue some type of output message via *printf* or *cprintf* from within the exit trap. In order for this to work, you should *fopen* or *open* the console device and send the message via *fprintf* or *write*.

SEE

exit, _exit

EXAMPLE

```
/*
*
* This program tests the "onexit" function.
*
*/
#include <stdlib.h>
#include <stdio.h>

int ex(i)     /* This is the exit trap function */
int i;
{
FILE *con;

if((con = fopen("*","w")) != NULL)
  {
  fprintf(con,"Exit trap hit...code %d found\n",i);
  fclose(con);
  }
return(0);
}

main()        /* This tests the exit trap */
{
int (*p)();
p = &ex;
printf("Setting exit trap...\n");
if(!onexit(p)) printf("Can't set trap...\n");
printf("Exiting with code 2\n");
exit(2);
}
```

NAME

open Open a Level 1 file

SYNOPSIS

```
#include <fcntl.h>

fh = open(name,mode,prot);

int fh;             file handle
char *name;         file name
int mode;           access mode
int prot;           protection mode (O_CREAT only)
```

DESCRIPTION

This function opens a file so that it can be accessed via the level 1 I/O functions. The file name can be any character string that is a valid file name, and it may include a device code and a directory path. The access mode is formed by ORing together the appropriate symbols from the following list of conventional unix symbols:

O_RDONLY Read-only access. No writes are allowed.

O_WRONLY Write-only access. No reads are allowed.

O_RDWR Read-write access. Both reads and writes are allowed.

O_NDELAY This symbol is defined for UNIX compatibility and has no effect under AmigaDOS.

O_APPEND This symbol is normally used in conjunction with O_WRONLY or O_RDWR. It causes the I/O system to seek to the end of the file before each write operation. After each write operation, the file is positioned at the new end-of-file.

O_TRUNC If the file exists, it is truncated to a length of 0. This flag is normally used with O_CREAT, O_WRONLY or O_RDWR.

O_CREAT	If the file does not already exist, it is created. The protection mode argument is provided for compatibility with existing software, but is ignored under AmigaDOS. To set the protection use the *chmod* function to change the protection bits after the file has been closed.
O_EXCL	This symbol is used only with O_CREAT. If O_EXCL and O_CREAT are both present and the file already exists, the *open* function will fail.

RETURNS

If the operation is successful, the function returns a file handle, which is an integer equal to or greater than 0. Otherwise it returns -1 and places error information in *errno* and *_OSERR*.

SEE

errno, _OSERR, chgfa, chmod, close, creat

NAME

perror Print UNIX error message

SYNOPSIS

```
#include <stdio.h>

error = perror(s);

int error;    contents of errno
char *s;      message prefix
```

DESCRIPTION

This function checks *errno* and, if it is non-zero, sends an error message to **stderr**. The message consists of the specified prefix, a colon and space, and the message text from the external array named **sys_errlist**. This array contains pointers to the various UNIX error messages. The highest error number is given by the contents of external integer **sys_nerr**. The Lattice compiler package contains the source for these two external items in a file named **syserr** which allows you modify the messages as you desire.

RETURNS

The function returns the contents of *errno* so you can test for an error condition and print a message in one step, as in the example:

```
if(perror("foo")) goto abort;
```

Note that this feature is an extension of UNIX and if you use it, your program may be non-portable.

SEE

errno, sys_nerr, sys_errlist, poserr

NAME

poserr Print AmigaDOS error message

SYNOPSIS

```
#include <dos.h>

error = poserr(s);

int error;     contents of _OSERR
char *s;       message prefix
```

DESCRIPTION

This function checks *_OSERR* and, if it is non-zero, sends an error message to **stderr**. The message consists of the specified prefix, a colon and space, and the message text from the external array named **os_errlist**. This array of structures contains pointers to the various AmigaDOS error messages. The highest error number is given by the contents of external integer **os_nerr**. The Lattice compiler package contains the source for these two external items in a file named **oserr.c** which you can modify to customize or expand the messages.

RETURNS

The function returns the contents of *_OSERR* so you can test for an error condition and print a message in one step, as in the example:

```
if(poserr("foo")) goto abort;
```

SEE

_OSERR, os_errlist, os_nerr, perror

NAME

putenv Put string into environment

SYNOPSIS

```
#include <stdlib.h>

error = putenv(env);

int error;      0 if successful
char *env;      environment string
```

DESCRIPTION

The *putenv* function accepts a string that has the form

```
name=var
```

and places it into the current environment. If the environment already contains a string beginning with *name*= then that string is replaced; otherwise, the new string is added.

This is accomplished by writing the text of **var** into the file *env:name*. If it is unable to write to the file, then a non-zero return code will be returned.

Environment variables on the Amiga are global so that writing a envirnoment from one process will set it for all processes.

SEE

getenv

EXAMPLE

```
#include <stdlib.h>

if(putenv("HOCUS=pocus"))  /* Add HOCUS */
```

```
    fprintf(stderr,"Couldn't add HOCUS\n");

  putenv("HOCUS=");          /* Remove HOCUS */
```

NAME

qsort	Sort a data array
dqsort	Sort an array of doubles
fqsort	Sort an array of floats
lqsort	Sort an array of long integers
sqsort	Sort an array of short integers
tqsort	Sort an array of text pointers

SYNOPSIS

```
#include <stdlib.h>

qsort(a,n,size,cmp);
dqsort(da,n);
fqsort(fa,n);
lqsort(la,n);
sqsort(sa,n);
tqsort(ta,n);

char *a;        data array pointer
double *da;     pointer to double array
float *fa;      pointer to float array
long *la;       pointer to long int array
short *sa;      pointer to short int array
char *ta[ ];    pointer to text pointer array

int n;          number of elements in array
int size;       element size in bytes
int (*cmp)();   pointer to comparison function
```

DESCRIPTION

The *qsort* function sorts the specified data array using the ACM 271 algorithm, more popularly known as *Quicksort*. During its operation, it calls upon the specified comparison routine with pointers to the two array elements being compared. As an extension to the normal UNIX implementation, we also pass a third argument, the element size, which can be ignored. The comparison routine should return an integral result as follows:

RETURN	MEANING
Negative	First element is below second
Positive	First element is above second
Zero	Elements are equal

The *dqsort*, *fqsort*, *lqsort*, *sqsort* and *tqsort* functions sort various arrays which are commonly encountered. They are all straightforward except for *tqsort*, which requires some explanation. The **ta** array consists of pointers to null-terminated character strings. The *tqsort* function re-arranges the pointers so that the strings are in ascending ASCII sequence, using *strcmp* as the comparison routine. Note that the sort is based on the contents of the strings rather than their physical address.

NAME

rand	Generate a random number
srand	Set seed for rand function

SYNOPSIS

```
#include <stdlib.h>

x = rand();
srand(seed);

int x;              random number
unsigned seed;      random number seed
```

DESCRIPTION

The *rand* function returns pseudo-random numbers in the range from 0 to the maximum positive integer value. When you call *srand*, the random number generator is reset to a new seed value. The initial default seed is 1.

See *drand48* and its related functions for more sophisticated random number generation.

RETURNS

As noted above.

SEE

drand48

EXAMPLE

```
/*
*
* This example prints 1000 random numbers
*
*/
#include <stdio.h>
```

Class: ANSI

```
#include <stdlib.h>

main(argc,argv)
int argc;
char *argv[ ];
{
int i;
unsigned x;

if(argc > 1)
  {
  stcd_u(argv[1],&x);
  if(x == 0) x = 1;
  printf("Seed value is %d\n",x);
  srand(x);
  }
printf("Here are 1000 random numbers...\n");
for(i = 0; i < 200; i++)
  printf("%5d %5d %5d %5d %5d\n",
         rand(),rand(),rand(),rand(),rand());
printf("\n\n");
}
```

NAME

read	Read from Level 1 file
write	Write to Level 1 file

SYNOPSIS

```
#include <fcntl.h>

count = read(fh,buffer,length);
count = write(fh,buffer,length);

unsigned int count;      actual bytes read or written
int fh;                  file handle
char *buffer;            data buffer
unsigned int length;     number of bytes to read or write
```

DESCRIPTION

These functions read or write a Level 1 file whose handle was returned by *creat* or *open*. Under normal circumstances, the value returned should match the buffer length. If this value is -1 or greater than the requested length, then some type of error occurred, and you should consult *errno* and *_OSERR*. If the actual length is less than the requested length when reading, this usually means that the file is exhausted. Similarly, if the actual length is less than the requested length for a write operation, this usually means that the device has no more space available. In both of these cases, it is still a good idea to check *errno* and *_OSERR* just in case some malfunction caused the short count.

Note that these functions are very similar to the Level 0 I/O functions *dread* and *dwrite*. The primary difference is that Level 1 files will be automatically closed by *exit* and *_exit*, which are usually called for you when the program terminates.

RETURNS

If the operation is successful, the function returns the actual number of bytes transferred. Otherwise it returns -1 and places error information in *errno* and *_OSERR*.

SEE

errno, _OSERR, open, dread, dwrite

NAME

remove	Remove a file
unlink	Remove a file

SYNOPSIS

```
#include <stdio.h>

error = remove(name);
error = unlink(name);

int error;          non-zero if error
char *name;         file name
```

DESCRIPTION

These functions remove the specified file from the system. They behave identically, but *unlink* is provided for compatibility with some versions of UNIX. The *remove* function is preferred because it is now in the ANSI C standard.

The file name argument can include a path, but it cannot include wild card characters. That is, you can remove only one file at a time.

RETURNS

If a non-zero value is returned, some type of error occurred, and additional information can be found in *errno* and *_OSERR*. The most common errors occur when you try to remove a file that doesn't exist or that is marked as read-only.

SEE

errno, _OSERR

EXAMPLE

This program removes all files specified in the argument list. It does not allow wild card characters in the file names.

```
#include <stdio.h>

main(argc,argv)
int argc;
char *argv[ ];
{
int i;           /* loop counter */
int ret = 0;     /* exit code, non-zero if any failures */

for(i = 1; i < argc; i++)
   if(remove(argv[i]))
  {
  perror("RMV");
  ret = 1;
  }
exit(ret);
}
```

NAME

rename Rename a file

SYNOPSIS

```
#include <fcntl.h>

error = rename(old,new);

int error;      0 for success, -1 for error
char *old;      old file name
char *new;      new file name
```

DESCRIPTION

This function renames a file, if possible. The old name can include a path, but the new name should not. A failure occurs if the old file cannot be found or if the new name is the same as an existing file.

RETURNS

If the function fails, it returns -1 and places additional error information into *errno* and *_OSERR*. Success is indicated by a return value of 0.

EXAMPLE

```
/*
*
* This is a version of the rename command
* that prompts for the old and new names.
*
*/
#include <stdlib.h>
#include <fcntl.h>
#include <dos.h>
```

```
main(argc,argv)
int argc;
char *argv[ ];
{
char old[FMSIZE],new[FMSIZE];
char *pold,*pnew;

if(argc < 2)    /* Get old file name */
  {
  printf("OLD FILE: ");
  if(gets(old) == NULL) exit(1);
  pold = old;
  }
else pold = argv[1];

if(arc < 3)
  {
  printf("NEW FILE: ");
  if(gets(new) == NULL) exit(1);
  pnew = new;
  }
else pnew = argv[2];

if(rename(pold,pnew))
  {
  perror("RENAME");
  exit(1);
  }
}
```

NAME

rlsmem	Release a Level 2 memory block
rlsml	Release a Level 2 memory block

SYNOPSIS

```
#include <stdlib.h>

error = rlsmem(p,sbytes);
error = rlsml(p,lbytes);

int error;          non-zero if error
char *p;            block pointer
short sbytes;       number of bytes
long lbytes;        number of bytes
```

DESCRIPTION

These functions release memory blocks that were previously obtained via *getmem* or *getml*. If the number of bytes is less than the value **MELTSIZE** as defined in header file **dos.h**, then it is made equal to **MELTSIZE**.

Note that you can free a portion of a block as long as the portion being freed is at least **MELTSIZE** bytes long. For example, suppose you allocate a block of 100 bytes, use the lower 80, and decide to return the upper 20 to the memory pool. Simply make the following call:

```
rlsmem(p+80,20);
```

where **p** is the original block pointer. Again, whenever you free partial blocks in this way, make sure that the portion being freed is equal to or greater than **MELTSIZE**.

RETURNS

If the block is not in the current Level 2 memory pool or overlaps a block that is already free, a value of -1 is returned. Otherwise, the return value is 0.

EXAMPLE

```
/*
*
* This program builds a linked list of text strings
* obtained from the standard input file.
*
*/
#include <stdio.h>
#include <stdlib.h>
/*
*
* These elements are linked together to form the
* text string list.
*
*/
#define MAX 256
struct LIST
  {
  struct LIST *next;     /* forward linkage */
  int size;              /* element size */
  char text[MAX];        /* minimum text string */
  };
/*
*
* Main program
*
*/
main()
{
struct LIST *p,*q,*list = NULL;
char b[MAX];
int x;
/*
*
* Build the list
*
*/
for(q = (struct LIST *)(&list);;q = p)
  {
  printf("Enter a text string: ");
```

```
    if(gets(b) == NULL) break;
    if(b[0] == '\0') break;
    p = (struct LIST *)getmem(sizeof(struct LIST));
    if(p == NULL)
      {
      printf("No more memory\n");
      break;
      }
    p->next = q->next;
    x = MAX - (stccpy(p->text,b,256);
    p->size = sizeof(struct LIST) - x;
    if(x >= MELTSIZE) rlsmem(p+p->size,x);
    }
  /*
  *
  * Print the list
  *
  */
  printf("\n\nTEXT LIST...\n");
  for(p = list; p != NULL; p = p->next)
     printf("%s",p->text);
  }
```

NAME

rmdir Remove a directory

SYNOPSIS

```
#include <stdio.h>

error = rmdir(path);

int error;        0 if successful
char *path;       points to directory path string
```

DESCRIPTION

This function removes an existing directory in the specified path. For example, if **path** is *sys:/abc/def/ghi*, then the directory named *ghi* is removed from the path */abc/def* on the system drive. For AmigaDOS, the path may begin with a drive or volume name and a colon.

RETURNS

If the operation is successful, the function returns 0. Otherwise it returns -1 and places error information in *errno* and *_OSERR*.

SEE

errno, _OSERR

NAME

rstmem Reset memory pool

SYNOPSIS

```
#include <stdlib.h>

rstmem();
```

DESCRIPTION

The *rstmem* function frees all memory allocated after the most recent *bldmem* call. *_pool* points to the first block of system memory to be released.

This capability is quite useful if you are in complete control of the memory pool. However, unless you are very careful, you can pull the rug out from under yourself by freeing memory that is still in use. For example, a file that has been opened for Level 2 I/O access (e.g. via the *fopen* function) will usually have a memory block allocated. Therefore, you should not call *rstmem* while Level 2 files are open.

RETURNS

None.

SEE

getmem, getml, rlsmem, rlsml, sizmem

NAME

setbuf	Set buffer mode for a Level 2 file
setnbf	Set non-buffer mode for L2 file
setvbuf	Set variable buffer for L2 file

SYNOPSIS

```
#include <stdio.h>

setbuf(fp,buff);
setnbf(fp);
error = setvbuf(fp,buff,type,size);

int error;      0 if successful
FILE *fp;       file pointer
char *buff;     buffer pointer
int type;       type of buffering
int size;       buffer size in bytes
```

DESCRIPTION

These functions set the buffering mode for a Level 2 file. Proper usage is to call the appropriate function after calling *fopen* and before calling any other Level 2 I/O functions. If you fail to follow this rule, the file may become corrupted.

The Level 2 I/O system automatically allocates a buffer via *getmem* when you perform the first read or write operation. Then the data being read or written is staged through this buffer in order to improve I/O efficiency. If you would rather use your own buffer instead of having one allocated for you, call *setbuf* with a non-null buffer pointer. The buffer size must be at least as large as the value given in the external integer *_bufsiz*, which defaults to the value of the symbol **BUFSIZ**, defined in **stdio.h**.

You can eliminate buffered I/O by calling *setnbf* or by calling *setbuf* with a null buffer pointer. When this is done, physical I/O occurs whenever your program performs Level 2 read or write operation, even if only one byte is

being transferred. This is very inefficient for disk files but often desirable for terminal or communication ports.

The *setvbuf* function can do everything that the other two functions can do, and it can also set *line buffered* mode and attach a buffer of non-standard size. The **type** argument must be one of the following symbols defined in **stdio.h**:

VALUE	MEANING
_IOFBF	Fully buffered
_IOLBF	Line buffered
_IONBF	Non-buffered

For **_IOFBF** and **_IOLBF**, the specified buffer will be attached to the file unless **buff** is null, in which case a buffer will be automatically allocated on the first read or write. For the **_IONBF** case, the **buff** and **size** arguments are ignored.

The line-buffered mode is useful for interactive applications. When in this mode, the buffer is flushed whenever a newline is sent, the buffer is full, or input is requested. Note, however, that you must use the *fputc* and *fputchar* functions instead of the *putc* and *putchar* macros in order for line buffering to work correctly. The macros do not check if line-buffered mode is active, and so they behave as if the file were fully buffered.

RETURNS

For *setvbuf*, the error code is non-zero if **type** or **size** is invalid.

> NOTE: These functions must be used only after *fopen* and before any other Level 2 file operations. A common error is to allocate a buffer on the stack within a function, attach it to a file, and then return from the function. This will corrupt the stack and cause a system failure.

SEE

fopen

NAME

signal Establish event traps

SYNOPSIS

```
#include <signal.h>

oldfun = signal(sig,newfun);

int (*oldfun)();     old trap function
int sig;             signal number
int (*newfun)();     new trap function
```

DESCRIPTION

This function establish traps for various events that can occur outside of your program.

The **newfun** argument specifies the action to be taken when the signal occurs, as follows:

SIG_IGN	Ignore the signal.
SIG_DFL	Take the system default action, as indicated above for each signal.

If **newfun** is not any of the above, then it must be a valid function pointer. When the signal is detected, the action is reset to either SIG_DFL or SIG_IGN, depending on the particular signal. Then the trap function is called with an integer argument specifying which signal was detected (e.g. SIGINT). The trap function can take whatever action is necessary, including calling **signal** again to re-establish itself as the trap function. If the function returns, execution continues at the point in your program where the signal was detected.

The **sig** argument specifies which signal is being trapped, using the following symbols defined in **signal.h**:

SIGFPE This signal occurs whenever a floating point error is detected and the standard version of **CXFERR** is installed. If you install your own version, you must duplicate our code (supplied as a file named *CXFERR.C*) in order to provide this signal.

SIGINT This signal occurs whenever the user operates the Ctrl-C or Ctrl-D key combination. The default action for AmigaDOS is to abort your program. If you specify a function to be called, the signal will be reset to SIG_IGN when the interrupt occurs. Your function should call **signal** again if you want to re-install the trap.

RETURNS

The **signal** function normally returns the previous value of the trap function, which may be SIG_IGN or SIG_DFL.

NAME

sizmem Get Level 2 memory pool size

SYNOPSIS

```
#include <stdlib.h>

size = sizmem();

long size;
```

DESCRIPTION

This function returns the number of unallocated bytes in the current Level 2 memory pool. This value is the sum of the sizes of all unallocated blocks, and so it does not indicate the size of the largest free block.

Also, the value does not indicate the maximum amount of Level 2 memory that can be allocated. That is, the Level 2 allocation functions *getmem* and *getml* will automatically expand the pool via *sbrk* when no block of sufficient size is found in the pool.

SEE

getmem, getml, rlsmem, rlsml, rstmem

NAME

stcarg Get an argument

SYNOPSIS

```
#include <string.h>

length = stcarg(s,b);

int length;   number of bytes in argument
char *s;      text string pointer
char *b;      break string pointer
```

DESCRIPTION

This function scans the text string until one of the break characters is found or until the null terminating byte is hit. While scanning, *stcarg* skips over substrings that are enclosed in single or double quotes, and the backslash is recognized as an escape character. In other words, break characters will not be detected if they are quoted or preceded by a backslash.

RETURNS

The function returns a count of the number of characters in **s** up to but not including the break character or null terminator.

SEE

stpbrk, strcspn, strpbrk

EXAMPLE

```
#include <stdio.h>
#include <string.h>

main()
{
char a[256],b[256];
int x;
```

```
while(1)
  {
  printf("Enter text string:  ");
  if(gets(a) == NULL) exit(0);
  printf("Enter break string: ");
  if(gets(b) == NULL) exit(0);
  x = stcarg(a,b);
  printf("Arg length: %d, Arg text: %.*s\n",x,x,a);
  }
}
```

NAME

stccpy	Copy one string to another
stpcpy	Copy one string to another
strcpy	Copy one string to another
strncpy	Copy string, length-limited

SYNOPSIS

```
#include <string.h>

size = stccpy(to,from,n)
np = stpcpy(to,from);
p = strcpy(to,from);
p = strncpy(to,from,n);

char *np;     points to end of destination string
char *p;      same as destination pointer
char *to;     destination pointer
char *from;   source pointer
int n;        maximum source length
```

DESCRIPTION

These functions copy the null-terminated source string to the destination area. For *stpcpy* and *strcpy*, the entire source string is copied, and the resulting destination is always null-terminated. The *strncpy* function always writes exactly **n** characters to the destination. If the null terminator is hit before **n** characters are copied from the source, then the destination is filled with null bytes. If the source string contains more than **n** non-null characters, the destination will not be null-terminated.

The *stccpy* function is similar to *strncpy* except that it always produces a null-terminated string, and it returns actual number of bytes placed in the **to** area, including the null terminator.

Note that *strcpy* has a built-in version which is functionally equivalent to the standard library version. The statement *#include <string.h>* provides a de-

fault setting by which built-in functions are accessed. If you don't want the built-in function, you can use an #undef statement as follows: #undef strcpy.

RETURNS

The *strcpy* and *strncpy* functions return a pointer that is the same as the destination pointer. The Lattice function *stpcpy* returns a pointer to the end of the destination string, which is often more useful when you are building a string up from several pieces.

> NOTE: Be careful when using *strncpy*, since it is one of the few string functions which does not produce a null-terminated string under every condition.

EXAMPLE

```
/*
*
* This example should print: Hello, my name is Flo.
*
*/
#include <string.h>

main()
{
char b[256],*p;

p = stpcpy(b,"Hello, ");
p = stpcpy(p,"my name is ");
p = stpcpy(p,"Flo.");
puts(b);
}
```

NAME

stcd_i	Convert decimal string to int
stco_i	Convert octal string to int
stch_i	Convert hexadecimal string to int
stcd_l	Convert decimal string to long int
stco_l	Convert octal string to long int
stch_l	Convert hexadecimal string to long

SYNOPSIS

```
#include <string.h>

length = stcd_i(in,ivalue);
length = stco_i(in,ivalue);
length = stch_i(in,ivalue);
length = stcd_l(in,lvalue);
length = stco_l(in,lvalue);
length = stch_l(in,lvalue);

int length;       input length
char *in;         input string pointer
int *ivalue;      integer value pointer
long *lvalue;     long integer value pointer
```

DESCRIPTION

These functions scan an input string and convert the leading characters into short or long integers. For *stcd_i* and *stcd_l*, the input string must begin with a plus sign '+', minus sign '-', or a decimal digit ('0' to '9'). The octal conversions *stco_i* and *stco_l* process an unsigned string of octal digits ('0' to '7'). Finally, the hexadecimal conversions *stch_i* and *stch_l* handle unsigned strings containing digits from '0' to '9' and letters from 'A' to 'F' or 'a' to 'f'. Scanning of the input string stops when the first invalid character is reached, At that point, the resulting value is stored into the area addressed by the second argument.

RETURNS

Each function returns the number of input characters converted. This result will be 0 if the first character of the input string is not valid for the particular conversion. In that case, conversion result stored via the second argument will be 0.

EXAMPLE

```
#include <stdio.h>
#include <string.h>

main()
{
int x;
long j;
char b[80];

while(1)
   {
   printf("\nEnter a hexadecimal value: ");
   if(gets(b) == NULL) exit(0);
   x = stch_l(b,&j);
   printf("stch_l: Length %d, Result %lx\n",x,j);
   }
}
```

NAME

stcgfe	Get file extension
stcgfn	Get file node
stcgfp	Get file path

SYNOPSIS

```
#include <string.h>

size = stcgfe(ext,name);
size = stcgfn(node,name);
size = stcgfp(path,name);
int size;       size of result string
char *ext;      extension area pointer
char *node;     node area pointer
char *path;     path area pointer
char *name;     file name pointer
```

DESCRIPTION

These functions isolate the path, node, or extension portion of a file name. The node is the rightmost portion of the file name that is separated from the rest of the name by a colon, slash, or backslash. The extension is the final part of the node that begins with a period, and the path is the leading part of the name up to the node. For example:

NAME	PATH	NODE	EXTENSION
"myprog.c"	""	"myprog.c"	"c"
"/abc.dir/def"	"abc.dir"	"def"	""
"/abc.dir/def.ghi"	"/abc.dir"	"def.ghi"	"ghi"
"df0:yourfile"	"df0:"	"yourfile"	""
"/abc/"	"/abc"	""	""

RETURNS

The **size** value is the same as would be returned by the *strlen* function. That is, if **size** is 0, then the desired portion of the file name could not be found and the result area contains a null string.

SEE

strsfn

EXAMPLE

```
#include <string.h>
#include <stdio.h>
main()
{
char file[FMSIZE],path[FMSIZE],node[FNSIZE],ext[FESIZE];

while(gets(file) != NULL)
  {
  stcgfe(ext,file);
  stcgfn(node,file);
  stcgfp(path,file);
  printf("PATH:\"%s\"  NODE: \"%s\"  EXT: \"%s\"\n",
    path,node,ext);
  }
}
```

NAME

stcis	Measure span of chars in set
stcisn	Measure span of chars not in set
strspn	Measure span of chars in set
strcspn	Measure span of chars not in set

SYNOPSIS

```
#include <string.h>

length = stcis(s,b);
length = stcisn(s,b);
length = strspn(s,b);
length = strcspn(s,b);

int length;    span length in bytes
char *s;       points to string being scanned
char *b;       points to character set string
```

DESCRIPTION

These functions measure the number of characters at the beginning of input string **s** that are either in or not in the character set specified by **b**. The *stcis* and *strspn* functions are identical and count the number of leading characters that are in the set. Similarly, *stcisn* and *strcspn* are identical and count the number of leading characters that are not in the set. The *stc* pair are provided for compatibility with other version of Lattice C, while the *str* functions are now part of the ANSI standard.

RETURNS

The functions all return the number of bytes that are in or not in the specified character set. Note that the scan always stops when the null terminator byte is reached.

EXAMPLE

```
#include <stdio.h>
#include <string.h>

main()
{
char s1[256],s2[256];

while(1)
        {
        printf("\nEnter test string: ");
        if(gets(s1) == NULL) exit(0);
        printf("Enter span string: ");
        if(gets(s2) == NULL) exit(0);
        printf("strspn: %d\n",strspn(s1,s2));
        printf("strcspn: %d\n",strcspn(s1,s2));
        printf("stcis: %d\n",stcis(s1,s2));
        printf("stcisn: %d\n",stcisn(s1,s2));
        }
}
```

NAME

stci_d	Convert int to decimal
stci_o	Convert int to octal
stci_h	Convert int to hexadecimal
stcl_d	Convert long int to decimal
stcl_o	Convert long int to octal
stcl_h	Convert long int to hexadecimal
stcu_d	Convert unsigned int to decimal
stcul_d	Convert unsigned long to decimal

SYNOPSIS

```
#include <string.h>

length = stci_d(out,ivalue);
length = stci_o(out,ivalue);
length = stci_h(out,ivalue);
length = stcl_d(out,lvalue);
length = stcl_o(out,lvalue);
length = stcl_h(out,lvalue);
length = stcu_d(out,uivalue);
length = stcul_d(out,ulvalue);

int length;                   output length
char *out;                    output buffer pointer
int ivalue;                   integer value
long lvalue;                  long integer value
unsigned int uivalue;         unsigned integer value
unsigned long ulvalue;        unsigned long integer value
```

DESCRIPTION

These functions convert various integral values into ASCII strings. The output area must be large enough to accomodate the maximum possible string, including the terminating null byte that each function appends. The following table shows the required lengths.

FUNCTION	LENGTH
stci_d	07
stci_o	07
stci_h	05
stcl_d	13
stcl_o	12
stcl_h	09
stcu_d	06
stcul_d	12

For *stci_d* and *stcl_d*, the first output character will be a minus sign if the input value is negative. No special leading character is generated if the value is positive. For all functions, leading zeroes are suppressed, and a single '0' character is generated if the input value is 0.

RETURNS

The return value is the number of characters actually placed into the output area, not including the final null byte.

EXAMPLE

```
#include <stdio.h>
#include <string.h>

main()
{
int i,x;
char b[13];

while(1)
        {
        printf("\nEnter a short integer: ");
        scanf("%d",&i);
        x = stci_d(b,i);
        printf("stci_d: Length %d, Result %s\n",x,b);
        x = stci_o(b,i);
        printf("stci_o: Length %d, Result %s\n",x,b);
```

```
        x = stci_h(b,i);
        printf("stci_h: Length %d, Result %s\n",x,b);
        }
}
```

NAME

stcpm	Un-anchored pattern match
stcpma	Anchored pattern match

SYNOPSIS

```
#include <string.h>

size = stcpm(string,pattern,match);
size = stcpma(string,pattern);

int size;              size of matching string
char *string;          string to be scanned
char *pattern;         pattern string
char **match;          returns pointer to matching string
```

DESCRIPTION

These functions scan a string to find a specified pattern. The pattern is specified in a simplified form of regular expression notation where

PATTERN	MEANING
?	matches any single character
c*	matches zero or more occurrences of character c
c+	matches one or more occurrences of character c
\?	matches a question mark (?)
*	matches an asterisk (*)
\+	matches a plus sign (+)

Any other character must match exactly. Some examples are:

PATTERN	MATCHES
"abc"	Only "abc"
"ab*c"	"ac", "abc" or "abbc" etc.
"ab+c"	"abc", "abbc" or "abbbc" etc.
"ab?*c"	A string starting with "ab" and ending in "c", e.g."abxyzc"
"ab*c"	Only "ab*c"

For *stcpma*, the match must occur at the beginning of the string, while for *stcpm*, the match can occur anywhere in the string. In either case, the function returns the size of the matching string or zero if there was no match. Also, *stcpm* returns a pointer to the beginning of the matching string.

EXAMPLE

```
#include <stdio.h>
#include <string.h>

main()
{
char s[100],p[100],*r;
int x;

while(1)
  {
  printf("\nSearch string => ");
  if(gets(s) == NULL) break;
  printf("Pattern       => ");
  if(gets(p) == NULL) break;
  x = stcpma(s,p);
  if(x)
    printf("stcpma: size = %d, match = \"%.*s\"\n",x,x,s);
  else
    printf("stcpma: no match\n"):
  x = stcpm(s,p,&r);
  if(x)
    printf("stcpm: size = %d, match = \"%.*s\"\n",x,x,r);
  else
    printf("stcpm: no match\n");
  }
}
```

NAME

stpblk Skip blanks (white space)

SYNOPSIS

```
#include <string.h>

q = stpblk(p);

char *q;    updated string pointer
char *p;    string pointer
```

DESCRIPTION

This function advances the string pointer past *white space* characters, that is, past all the characters for which *isspace* is true.

RETURNS

The function returns a pointer to the next non *white space* character. Note that the null terminator byte is not considered to be white space, and so the function will not go past the end of the string.

SEE

stcis, strspn

EXAMPLE

```
#include <stdio.h>
#include <string.h>

main()
{
char input[256];
```

```
while(1)
    {
    puts("\nEnter a string with leading blanks...");
    if(gets(input) == NULL) exit(0);
    printf("%s\n",stpblk(input));
    }
}
```

NAME

stpbrk	Find break character in string
strpbrk	Find break character in string

SYNOPSIS

```
#include <string.h>

p = stpbrk(s,b);
p = strpbrk(s,b);

char *p;     points to break character in s
char *s;     string to be scanned
char *b;     break characters
```

DESCRIPTION

These functions scan string **s** to find the first occurrence of a character from break string **b**. They are completely equivalent, except that *strpbrk* s the ANSI name, while *stpbrk* is the traditional Lattice name.

RETURNS

If no character from **b** is found in **s**, a NULL pointer is returned. Otherwise, **p** is a pointer to the break first break character.

SEE

strspn, strcspn

EXAMPLE

```
#include <string.h>
#include <stdio.h>

/*
*
* Scan for commas, periods, and blanks.  Display
* the tail of the string each time a break
```

```
 * character is found.
 *
 */
char *p,s[ ] = "Hello, I must be going.";

for(p = s; (p = strbrk(p,",. ")) != NULL;)
       printf("%s\n",p);
```

NAME

stpchr	Find character in string
stpchrn	Find character not in string
strchr	Find character in string
strrchr	Find character not in string

SYNOPSIS

```
#include <string.h>

p = stpchr(s,c);
p = stpchrn(s,c);
p = strchr(s,c);
p = strrchr(s,c);

char *p;    updated string pointer
char *s;    input string pointer
char c;     character to be located
```

DESCRIPTION

The *stpchr* and *strchr* functions scan the input string to find the first occurrence of the character specified by argument **c**. Similarly, *stpchrn* and *strrchr* scan for the first occurrence of some character other than **c**. The *stp* versions are provided for compatibility with other versions of Lattice C, while the *str* functions are now part of the ANSI standard.

RETURNS

For *stpchr* and *strchr*, a NULL pointer is returned if the input string is empty or if the specified character is not found. The other two functions return a NULL pointer if the input string is empty or consists entirely of character **c**.

EXAMPLE

```
#include <stdio.h>
#include <string.h>
```

```
main()
{
char c,s[256];

while(1)
        {
        printf("\nEnter test string: ");
        if(gets(s1) == NULL) exit(0);
        printf("Enter character: ");
        if((c = getchar()) == EOF) exit(0);
        printf("stpchr: %s\n",stpchr(s,c));
        printf("stpchrn: %s\n",stpchrn(s,c));
        printf("strchr: %s\n",strchr(s,c));
        printf("strrchr: %s\n",strrchr(s,c));
        }
}
```

NAME

stpdate Convert date array to string

SYNOPSIS

```
#include <string.h>

np = stpdate(p,mode,date);

char *np;     updated output string pointer
char *p;      output string pointer
int mode;     conversion mode
char *date;   date array, as follows
              date[0] => year - 1980
              date[1] => month (1 to 12)
              date[2] => day (1 to 31)
```

DESCRIPTION

This function converts a 3-byte date array into ASCII or BCD according to the **mode** argument:

MODE	DATE FORMAT
0	yymmdd (BCD, 3 bytes)
1	yymmdd (ASCII, 7 bytes)
2	mm/dd/yy (ASCII, 9 bytes)
3	mm-dd-yy (ASCII, 9 bytes)
4	MMM d, yyyy (ASCII, up to 13 bytes)
5	Mm...m d, yyyy (ASCII, up to 19 bytes)
6	dd MMM yy (ASCII, 10 bytes)
7	dd MMM yyyy (ASCII, 12 bytes)

In the above formats, *MMM* represents a 3-character month abbreviation in capitals, and *Mm...m* represents the full month name (e.g. January). The *mm, dd* and *yy* terms are 2-character month, day, and year, respectively, while

d is the date with the leading zero suppressed. The *yyyy* term is the 4-character year obtained by adding 1980 to the first byte of the date array.

For all modes except 0, a null byte is appended to the output string.

RETURNS

The function does not make validity checks on the date array thus it cannot fail. It returns a pointer to the first byte past the generated output. For modes other than 0, this is a pointer to the null terminator.

SEE

stptime, getclk, getft

NAME

stpsym Get next symbol from a string

SYNOPSIS

```
#include <string.h>

p = stpsym(s,sym,symlen);

char *p;        points to next input character
char *s;        input string
char *sym;      output string
int symlen;     sizeof(sym)
```

DESCRIPTION

This function breaks out the next symbol from the input string. The first character of the symbol must be alphabetic (upper or lower case), and the remaining characters must be alphanumeric. Note that the pointer is not advanced past any initial white space in the input string.

The output string is the null-terminated symbol, and will be an empty string if no symbol is found. If the symbol is longer than **symlen-1**, its excess characters are dropped.

RETURNS

The function returns a pointer to the next character past the symbol.

SEE

stcarg, stpbrk, strcspn, strpbrk

EXAMPLE

```
#include <stdio.h>
#include <string.h>

main()
```

```
{
char a[256],b[10];

while(1)
  {
  printf("\nEnter text string:  ");
  if(gets(a) == NULL) exit(0);
  While(1)
    {
    p = stpsym(a,b,sizeof(b));
    printf("Symbol: \"%s\"  Residual: \"%s\"\n",b,p);
    if(b[0] == '\0') break;
    }
  }
}
```

NAME

stptime Convert time array to string

SYNOPSIS

```
#include <string.h>

np = stptime(p,mode,time);

char *np;       updated output string pointer
char *p;        output string pointer
int mode;       conversion mode
char *time;     time array, as follows
                time[0] => hour (0 to 23)
                time[1] => minute (0 to 59)
                time[2] => second (0 to 59)
                time[3] => hundredths (0 to 99)
```

DESCRIPTION

This function converts a 4-byte time array into ASCII or BCD according to the **mode** argument:

MODE	TIME FORMAT
0	hhmmssdd (BCD, 4 bytes)
1	hhmmss (ASCII, 7 bytes)
2	hh:mm:ss (ASCII, 9 bytes)
3	hhmmssdd (ASCII, 9 bytes)
4	hh:mm:ss.dd (ASCII, 12 bytes)
5	hh:mm (ASCII, 6 bytes)
6	hr:mm:ss HH (ASCII, 12 bytes)
7	hr:mm HH (ASCII, 9 bytes)

The *hh, mm, ss* and *dd* terms are simply the 2-digit (BCD or ASCII) equivalents of the binary values in the time array. The *hr* term is the 2-digit hour using the 12-hour form, and the *HH* term is either AM or PM.

Note that a null terminator is appended to the ASCII output strings.

RETURNS

The function does not make validity checks on the time array, thus it cannot fail. It returns a pointer to the first byte past the generated output. For modes other than 0, this is a pointer to the null terminator.

SEE

stpdate, getclk, getft

NAME

stptok Get next token from a string

SYNOPSIS

```
#include <string.h>

p = stptok(s,tok,toklen,brk);

char *p;        points to next character after token
char *s;        points to input string
char *tok;      points to output buffer
int toklen;     sizeof(tok)
char *brk;      break string
```

DESCRIPTION

This function breaks out the next token from the input string and moves it to the token buffer with a null terminator. A token consists of all characters in the input string **s** up to but not including the first character that is in the break string. In other words, **brk** specifies the characters that cannot be included in a token.

If the input string begins with a break character, then the token buffer will contain a null string, and the return pointer **p** will be the same as **s**. If no break character is found after **toklen-1** input characters have been moved to the token buffer, or if the input string terminator (a null byte) is hit, then the scan stops as if a break character were hit.

RETURNS

The function returns a pointer to the next character in the input string.

> NOTE: This function does not delete white space at the beginning of the input string.

SEE

stpblk, strtok

EXAMPLE

```
/*
*
* This example breaks out words that are
* separated by blanks or commas.
* The token buffer takes on the following
* values as the program loops:
*
*    LOOP   TOKEN
*     1     first
*     2     second
*     3     third
*     4     fourth
*
*/
#include <string.h>
#include <stdio.h>

char test[ ] = "first, second third, fourth";

main()
{
char *p = test;
char token[50];

while(1)
        {
        p = stptok(p,token,sizeof(token)," ,");
        printf("%s\n",token);
        if(*p == '\0') break;
        p = stpblk(++p);
        }
}
```

NAME

strbpl Build string pointer list

SYNOPSIS

```
#include <string.h>

n = strbpl(s,max,t);

int n;          number of pointers
char *s[];      pointer to string pointer list
int max;        maximum number of pointers
char *t;        text pointer
```

DESCRIPTION

This function constructs a list of pointers to the strings contained within the specified text array. Each string must be null-terminated, and the text array must be terminated by a null string. In other words, array **t** must end with two null bytes, one to terminate the final string and another to terminate the array. The string pointer list **s** is terminated by a null pointer.

RETURNS

The return value indicates how many string pointers were placed into array **s**. If the number of strings plus the final null pointer is greater than **max**, a value of -1 is returned.

SEE

getfnl, strsrt

EXAMPLE

```
#include <string.h>

main( )
{
char text[ ] = {"string 1","string 2",'\0'};
```

```
char *list[5];
int count;

/*
* The following call has the following effect:
*
*    Return value (count) is 2.
*    list[0] => "string 1"
*    list[1] => "string 2"
*    list[2] => NULL
*
*/
count = strbpl(list,5,text);
}
```

NAME

strcat	Concatenate strings
strncat	Concatenate strings, max length

SYNOPSIS

```
#include <string.h>

p = strcat(to,from);
p = strncat(to,from,n);

char *p;        same as destination string pointer
char *to;       destination string pointer
char *from;     source string pointer
int n;          maximum source length
```

DESCRIPTION

These functions concatenate the source string to the tail end of the destination string. For *strncat*, no more than **n** characters are moved from the source to the destination. A null byte is placed at the end of the destination in any case.

RETURNS

Both functions return a pointer that is the same as the first argument.

SEE

stpcpy

EXAMPLE

```
#include <stdio.h>
#include <string.h>

main()
{
char a[256],b[256];
```

```
int n;

while(1)
    {
    printf("\nEnter string A: ");
    if(gets(a) == NULL) exit(0);
    printf("Enter string B: ");
    if(gets(b) == NULL) exit(0);
    printf("Enter maximum length N: ");
    scanf("%d",&n);
    printf("strcat(A,B):     \"%s\"\n",strcat(a,b));
    printf("strncat(A,B,N): \"%s\"\n",strncat(a,b,n));
    }
}
```

NAME

strcmp	Compare strings
strcmpi	Compare strings, case-insensitive
stricmp	Compare strings, case-insensitive
strncmp	Compare strings, length-limited
strnicmp	Compare strings, no case, max size

SYNOPSIS

```
#include <string.h>

x = strcmp(a,b);
x = strcmpi(a,b);
x = stricmp(a,b);
x = strncmp(a,b,n);
x = strnicmp(a,b,n);

int x;          comparison result
char *a,*b;     strings being compared
unsigned int n;
```

DESCRIPTION

These functions compare two null-terminated strings. The ASCII collating sequence is used in all cases, but *strcmpi*, *stricmp* and *strnicmp* do not distinguish between upper and lower case. Note that *strcmpi* is exactly the same as *stricmp*, except that the former is a hold-over from various Microsoft compilers, while the latter is in the ANSI standard.

The relative collating sequence of the strings is indicated by the sign of the return value, as follows:

RETURN	MEANING
Negative	First string is below second
Zero	Strings are equal
Positive	First string is above second

If the strings have different lengths, the shorter one is treated as if it were extended with zeroes. For *strncmp* and *strnicmp*, no more than **n** characters are compared.

Note that *strcmp* has a built-in version which is functionally equivalent to the standard library version. The statement *#include <string.h>* provides a default setting by which built-in functions are accessed. If you don't want the built-in function, you can use an #undef statement as follows: #undef strcmp.

RETURNS

As noted above.

EXAMPLE

```
#include <stdio.h>
#include <string.h>

main()
{
char a[256],b[256];
int n;

while(1)
  {
  printf("Enter string A: ");
  if(gets(a) == NULL) exit(0);
  printf("Enter string B: ");
  if(gets(b) == NULL) exit(0);
  printf("Enter maximum compare length: ");
  scanf("%d",&n);

  result("strcmp:   ",strcmp(a,b));
  result("stricmp: ",stricmp(a,b));
  result("strncmp: ",strncmp(a,b,n));
  result("strnicmp:",strnicmp(a,b,n));
  }
}

void result(name,r)
```

```
char *name;
int r;
{
char *p;

if(r == 0) p = "is equal to";
if(r < 0) p = "is less than";
if(r > 0) p = "is greater than";
printf("%s String A %s string B\n",name,p);
}
```

NAME

strdup Duplicate a string

SYNOPSIS

```
#include <string.h>

p = strdup(s);

char *p;     points to duplicate string
char *s;     points to string being duplicated
```

DESCRIPTION

This function creates a duplicate of the specified string by using *malloc* and *strcpy* to allocate space and copy the string to it.

RETURNS

A null pointer is returned if *malloc* fails. Otherwise, the function returns a pointer to the duplicate string.

NAME

strins Insert a string

SYNOPSIS

```
#include <string.h>

void strins(to,from);

char *to;       destination string
char *from;     source string
```

DESCRIPTION

This function inserts the source string in front of the destination. Both strings must be null-terminated, and the destination is shifted to the right (upward in memory) in order to accomodate the source string. The final result is a single null-terminated string.

NOTE: Ensure that the destination area is large enough, otherwise you could have a systems failure.

SEE

strcat

EXAMPLE

```
#include <stdio.h>
#include <string.h>

main( )
{
char here[] = "Here ";
char now[30] = "and now";
printf("%s, %s\n",here,now);
strins(now,here);          /* now => "Here and now" */
```

```
printf("%s\n",now);
}
```

NAME

strlen	Measure length of a string
stclen	Measure length of a string

SYNOPSIS

```
length = strlen(s);
length = stclen(s);
char *s;
int length;          number of bytes in s (before null)
```

DESCRIPTION

These functions return the number of bytes in string s before the null terminator byte. The *strlen* function is the ANSI equivalent of the Lattice implementation *stclen*.

Note that *strlen* has a built-in version which is functionally equivalent to the standard library version. The statement *#include <string.h>* provides a default setting by which built-in functions are accessed. If you don't want the built-in function, you can use an #undef statement as follows: #undef strlen.

RETURNS

length = number of bytes in string before null byte

EXAMPLE

```
x = strlen("abc");   /* x is 3 */
x = strlen("");      /* x is 0 */
```

NAME

strlwr	Convert string to lower case
strupr	Convert string to upper case

SYNOPSIS

```
#include <string.h>

p = strlwr(s);
p = strupr(s);

char *p;     return pointer (same as s)
char *s;     string pointer
```

DESCRIPTION

These functions convert all alphabetic characters in the specified null-terminated string to lower or upper case. In each case, the function return value is the same as the string pointer.

RETURNS

Both functions return the original string pointer.

NAME

strmfe Make file name with extension

SYNOPSIS

```
#include <string.h>

strmfe(newname,oldname,ext);

char *newname;    new file name
char *oldname;    old file name
char *ext;        extension
```

DESCRIPTION

This function copies the old file name to the new name, deleting any extension. Then it appends the specified extension to the new file name, with an intervening period. For example:

OLDNAME	EXT	NEWNAME
"df1:myprog.c"	"cc"	"df1:myprog.cc"
"abc"	"o"	"abc.o"

SEE

strmfn, strmfp

> NOTE: The **newname** area must be large enough to accept the file name string and the separator. A safe size is **FMSIZE**, which is defined in the **dos.h** header file.

NAME

strmfn   Make file name from components

SYNOPSIS

```
#include <string.h>

strmfn(file,drive,path,node,ext)

char *file;       file name pointer
char *drive;      drive code pointer
char *path;       directory path pointer
char *node;       node pointer
char *ext;        extension pointer
```

DESCRIPTION

This function makes a file name from four possible components. In general, the name is constructed as follows:

```
drive:path/node.ext
```

If the **drive** pointer is not null, that string is moved to the area pointed to by the **file** argument. Then a colon is inserted unless one is already there. Next, if **path** is not NULL, it is appended to **file**, and the directory separator specified by **_SLASH** is added if necessary. The **node** string is appended next, unless it is NULL. Finally, if **ext** is not NULL, a period is appended to **file**, followed by the **ext** string.

RETURNS

None

NOTE: Ensure that the **file** pointer refers to an area which is large enough to hold the result. A safe value is **FMSIZE**, which is defined in **dos.h**.

SEE

strmfe, strmfp, _SLASH

EXAMPLE

```
#include <dos.h>
#include <stdio.h>
#include <string.h>

char buffer[FMSIZE];

/* The next statements both place "abc/def/ghi" into the
   buffer. */


strmfn(buffer,NULL,"abc/def","ghi",NULL);
strmfn(buffer,NULL,"abc/def/","ghi",NULL);


/* The next statements both generate
   "df0:myfile.str"  */

strmfn(buffer,"df0",NULL,"myfile","str");
strmfn(buffer,"df0:",NULL,"myfile","str");
```

NAME

strmfp Make file name from path/node

SYNOPSIS

```
#include <string.h>

strmfp(name,path,node);

char *name;    file name
char *path;    directory path
char *node;    node
```

DESCRIPTION

This function copies the path string to the file name area, appending the **_SLASH** separator if the path string is not empty and does not end with a slash, backslash, or colon. Then the node string is appended to the file name. **_SLASH** is an external character variable that defaults to a slash (/).

> NOTE: The **name** area must be large enough to accept the file name string. A safe value is **FMSIZE**, which is defined in the *dos.h* header file.

SEE

strmfe, strmfn, _SLASH

NAME

strmid Return a substring from a string

SYNOPSIS

```
#include <string.h>

error = strmid(source,dest,pos,len);

char *dest,*source;   /* source and destination pointers */
int pos;              /* starting position of dest in source
*/
int len;              /* length of substring */
int error;            /* -1 if pos is beyond source,
                         else 0 */
```

DESCRIPTION

The *strmid* function returns a pointer to a substring of *source* beginning at character position *pos*, and having a length of *len*. If *len* is greater than the length of source offset at *pos*, then the rest of the string is copied to *dest*.

The destination string is null-terminated.

RETURNS

If *pos* is beyond the length of *source*, then -1 is returned. Otherwise, 0 is returned.

SEE

NAME

strnset	Set string to value, max length
strset	Set string to value

SYNOPSIS

```
#include <string.h>

p = strnset(s,c,n);
p = strset(s,c);

char *p;     return pointer (same as s)
char *s;     string pointer
int c;       value
int n;       maximum string length
```

DESCRIPTION

These functions set all bytes of a null-terminated string to the same value, not including the terminator byte. For *strnset* no more than **n** bytes will be set. That is, if a null byte is not found by the time **n** bytes have been set, the operation stops.

RETURNS

Both functions return the original string pointer.

NAME

strrev Reverse a character string

SYNOPSIS

```
#include <string.h>

p = strrev(s);

char *p,*s;    string pointer
```

DESCRIPTION

This function reverses a character string. In other words it *rotates* the string about its mid-point such that the last character is first and the first is last.

RETURNS

Returns same pointer that was passed to it.

NAME

strsfn Split the file name

SYNOPSIS

```
#include <string.h>

strsfn(file,drive,path,node,ext);

char *file;    file name pointer
char *drive;   drive code pointer
char *path;    directory path pointer
char *node;    node pointer
char *ext;     extension pointer
```

DESCRIPTION

This function splits a file name into four possible components and places them into the **drive**, **path**, **node** and **ext** strings. If any of those arguments are NULL, then those components are discarded.

In general, a complete file name is constructed as follows:

```
drive:path/node.ext
```

When *strsfn* splits the file name, it leaves the colon attached to the drive code, but drops the other punctuation characters. In other words, the **path** component will not end with a slash, and the **ext** component will not begin with a period. Slashes or backslashes within the **path** component are preserved.

> NOTE: You must make sure that the **drive**, **path**, **node** and **ext** pointer refer to areas that are large enough to hold the largest string that might be generated. Note that *strsfn* does not check that any component lengths are exceeded, although it does copy **file** string to an internal buffer of size FMSIZE and truncate it if it is too long. If you want to be absolutely sure that no overflows occur, make each component area be FMSIZE bytes long.

RETURNS

None

SEE

strgfn, strmfe, strmfn

EXAMPLE

```
#include <dos.h>
#include <stdio.h>
#include <stdlib.h>

char a[FNSIZE],b[FMSIZE],c[FNSIZE],d[FESIZE];

/* After the next statement, the component strings are:

        a => ""
        b => "abc/def"
        c => "ghi"
        d => ""

*/

strsfn("abc/def/ghi",a,b,c,d);

/* After the next statement, the component strings are:

        a => "df0:"
```

```
        b => ""
        c => "myfile"
        d => "str"

*/

strsfn("df0:myfile.str",a,b,c,d);

/*
*
* This program splits file names that it reads
* from stdin.
*
*/
#include <stdio.h>
#include <string.h>
#include <dos.h>

main()
{
char
input[255],drive[FNSIZE],path[FMSIZE],node[FNSIZE],ext[FESIZE];

while(1)
        {
        puts("\nEnter file name...\n");
        if(gets(input) == NULL) exit(0);
        strsfn(input,drive,path,node,ext);
        printf("DRIVE: \"%s\"\n",drive);
        printf("PATH:  \"%s\"\n",path);
        printf("NODE:  \"%s\"\n",node);
        printf("EXT:   \"%s\"\n",ext);
        }
}
```

NAME

strsrt Sort string pointer list

SYNOPSIS

```
#include <string.h>

strsrt(s,n);

char *s[];      string pointer list
int n;          number of pointers in list
```

DESCRIPTION

This function performs a simple bubble sort of the string pointers in the specified list. It is particularly useful in conjunction with the *getfnl* and *strbpl* functions. For large lists, you will usually get better performance using *tqsort*.

RETURNS

None.

SEE

getfnl, strbpl, tqsort

EXAMPLE

```
/*
/* This program constructs an array of pointers to all file
/* names in the current directory that have an extension of
/* ".c". Then the array is sorted into ASCII order.
*
*/

#include <string.h>

main( )
{
```

```
char names[3000],*pointers[300];
int count;

count = getfnl("#?.c",names,sizeof(names),0);
if(count > 0)
   {
   if(strbpl(pointers,300,names) != count) goto error;
   strsrt(pointers,count);
   }
}
```

NAME

strtok Get a token

SYNOPSIS

```
#include <string.h>

t = strtok(s,b);

char *t;    token pointer
char *s;    input string pointer or NULL
char *b;    break character string pointer
```

DESCRIPTION

This function treats the input string as a series of one or more tokens separated by one or more characters from the break string. By making a sequence of calls to *strtok*, you can obtain the tokens in left-to-right order. To get the first (leftmost) token, supply a non-NULL pointer for the s argument. Then to get the next tokens, call the function repeatedly with a NULL pointer for s, until you get a NULL return pointer to indicate that there are no more tokens. The break string can be changed from one call to another.

Each time it is entered, *strtok* takes the following steps:

1. If the input string is NULL, obtain the string pointer that was used on the preceding call. Otherwise use the new input string pointer.
2. Scan forward through the string to the next non-break character. If it is a null byte, return a value of NULL to indicate that there are no more tokens.
3. Scan forward through the string to the next break character or the null terminator. In the former case, write a null byte into the string to terminate the token, and then scan forward until the next non-break is found. In either case, save the final value of the string pointer for the next call, and return the token pointer.

Note that the input string gets changed as the scan progresses. Specifically, a null byte is written at the end of each token.

RETURNS

A NULL pointer is returned when there are no more tokens.

SEE

stptok, strcspn, strspn

EXAMPLE

```
/*
* This example breaks out words that are separated
* by blanks or commas. The token pointer takes on
* the following values as the program loops:
*
*   LOOP   TOKEN
*    1     "first"
*    2     "second"
*    3     "third"
*    4     "fourth"
*    5     NULL
*
*/
#include <string.h>
#include <stdio.h>

main ( )
{
char test[ ] = "first, second third, fourth";

char *token;
token = strtok(test,", ");
while(token != NULL)
  {
  printf("%s\n",token);
  token = strtok(NULL,", ");
  }
}
```

NAME

strtol Convert string to long integer

SYNOPSIS

```
#include <string.h>

r = strtol(p,np,base);

long r;       result
char *p;      input string pointer
char **np;    receives new input string pointer
int base;     conversion base
```

DESCRIPTION

This function converts an ASCII input string into a long integer according to the specified base, which can range from 0 to 36, excluding 1. Valid digit characters are 0 to 9, a to z, and A to Z. The highest allowable character is determined by the conversion base. For example, if the base is 17, then the string can contain digits from 0 to 9, a to g, and A to G.

The function skips leading white space and then checks for a leading plus or minus sign. In the latter case, the result of the conversion is negated before it is returned. The conversion stops at the first invalid character, and a pointer to that character is returned in **np** if the **np** argument is not NULL. Note that if the entire string is converted, **np** will contain a pointer to the null terminator byte.

If **base** is 0, then the string is analyzed to see if it is octal, decimal, or hexadecimal, as follows:

Base 16 If the string begins with 0x or 0X, base 16 (hexadecimal) conversion is performed.

Base 8 Otherwise, if the string begins with 0, base 8 (octal) conversion is performed.

Base 10 If neither of the above applies, base 10 (decimal) conversion is performed.

NOTE: There is no check is made for overflow.

SEE

atol, stcd_l

EXAMPLE

```
/*
* This program tests the strtol function.
*/
#include <stdio.h>
#include <string.h>

main()
{
char *p,buff[80];
int base;
long x;

while(1)
        {
        printf("\nEnter number base (0 to 36): ");
        if(gets(buff) == NULL) exit(0);
        if(buff[0] == '\0') exit(0);
        base = atoi(buff);
        if((base < 0) || (base > 36)) continue;

        printf("Enter number: ");
        if(gets(buff) == NULL) exit(0);
        if(buff[0] == '\0') exit(0);
        x = strtol(buff,&p,base);
        printf("Decimal result = %ld\n",x);
        if(*p != '\0') printf("Residual = %s\n",p);
        }
}
```

NAME

stspfp Parse file path

SYNOPSIS

```
#include <string.h>

error = stspfp(path,nx);

int error;     -1 for error, 0 for success
char *path;    file path string
int nx[16];    node index array
```

DESCRIPTION

This function parses a file path, which is a null-terminated string consisting of nodes separated by the **_SLASH** character. Each separator is replaced with a null byte, and the index to the first character of that node is placed into the node index array. The last entry in the array is followed by a -1. A leading separator in the path string is skipped.

RETURNS

A return value of -1 indicates that the path contains more than 15 nodes.

SEE

stcgfe, stcgfn, stcgfp, strsfn

EXAMPLE

```
/*
* The following piece of code parses /ABC/DE/F
* into strings ABC, DE, and F. The node index
* array will then contain 1, 5, 8, and -1.
*/
int xx[16];

stspfp("/ABC/DE/F",xx);
```

NAME

stub Default routine for undefined routines

SYNOPSIS

```
stub();
```

DESCRIPTION

The *stub* function is the default routine resolved by Blink for routines not found in libraries. By default, it will put up a requester indicating that the unwritten routine had been called. It is intended to allow development and testing of a program for which some of the routines have not been written (and, of course, are not expected to be called).

RETURNS

None.

NAME

system   Call system command processor

SYNOPSIS

```
#include <stdlib.h>

error = system(cmd);

int error;    non-zero if error
char *cmd;    command string
```

DESCRIPTION

This function invokes the system command processor and passes the **cmd** string to it. Under AmigaDOS, this function invokes the AmigaDOS **Execute** facility.

RETURNS

If the command processor cannot be invoked, a value of -1 is returned, and additional error information can be found in *errno* and *_OSERR*. Otherwise, the function returns the value passed back by the command processor.

SEE

errno, _OSERR

EXAMPLE

```
#include <stdlib.h>

main( )
{
int x;
x = system("copy #? to df1:");
if(x < 0) printf("System command failed\n");
if(x > 0) printf("System command error %d\n",x);
}
```

NAME

time — Get system time in seconds

SYNOPSIS

```
#include <time.h>

timeval = time(timeptr);

long timeval;     time value
long *timeptr;    pointer to time value storage
```

DESCRIPTION

This function returns the current time expressed as the number of seconds since 00:00:00 Greenwich Mean Time, January 1, 1970. If **timeptr** is not NULL, the time value is also stored in that location.

SEE

asctime, ctime, gmtime, localtime, tzset

EXAMPLE

```
#include <time.h>
#include <stdio.h>

main()
{
long t;

time(&t);
printf("Current time is %s\n",ctime(&t));
}
```

NAME

timer Get system clock with microseconds

SYNOPSIS

```
timer(clock);

unsigned int clock[2];
```

DESCRIPTION

The *timer* function obtains the current setting of the system clock into a 2-integer array as follows:

```
clock[0] => seconds
clock[1] => microseconds
```

RETURNS

None.

Class: ANSI

NAME

toascii	Convert character to ASCII
tolower	Convert character to lower case
toupper	Convert character to upper case

SYNOPSIS

```
#include <ctype.h>

cc = toascii(c);
cc = tolower(c);
cc = toupper(c);

int cc;    converted character
int c;     character to convert
```

DESCRIPTION

These functions convert characters into different forms. If you include **ctype.h** as shown above, they are actually defined as macros and produce in-line code to perform the conversions. Without **ctype.h**, they are actual functions resolved in the standard library. If you want to use the function version but must include **ctype.h** for some other reason, use **#undef** to undefine the conversion macros.

The *toascii* conversion simply resets all high-order bits, leaving only the lower seven. The *tolower* conversion tests if **c** is an upper case alphabetic character and, if so, converts it to lower case. Otherwise, **cc** is the same as **c**. The *toupper* conversion is the reverse of *tolower*.

SEE

_ctype

EXAMPLE

```
/*
*
* The following program echoes each input
* line in upper case.
*
*/
#include <stdio.h>
#include <ctype.h>

main()
{
char b[100],*p;

while(gets(b) != NULL)
        {
        for(p = b; *p != '\0'; p++) *p = toupper(*p);
        puts(b);
        }
}
```

NAME

MathTranBase FFP Library Vector

SYNOPSIS

```
extern long MathTransBase;
MathTransBase = OpenLibrary("mathtrans.library",ver);
```

DESCRIPTION

This external location is used to interface with the Motorola Fast Floating Point format transcendental function library routines provided by Amiga. It is initialized by an **OpenLibrary** call when a program compiled with the FFP option performs the first floating point transcendental function call. It contains the base address of the FFP transcendental math library vector table. If you make direct calls to the Amiga FFP transcendental functions you must first initialize this location by calling **OpenLibrary**.

NAME

cos	Cosine function
sin	Sine function
tan	Tangent function
cot	Cotangent function
acos	Arccosine function
asin	Arcsine function
atan	Arctangent function
atan2	Arctangent of x/y
cosh	Hyperbolic cosine function
sinh	Hyperbolic sine function
tanh	Hyperbolic tangent function

SYNOPSIS

```
#include <math.h>

r = cos(x);
r = sin(x);
r = tan(x);
r = cot(x);

r = acos(x);
r = asin(x);
r = atan(x);
r = atan2(x,y);

r = cosh(x);
r = sinh(x);
r = tanh(x);

double r;      result;
double x,y;    arguments
```

DESCRIPTION

The *cos, sin* and *tan* routines compute the normal circular functions of angles expressed in radians.

The *acos, asin, atan* and *atan2* routines compute the inverse circular functions, returning angular values expressed in radians. Results are constrained as follows:

FUNCTION	RETURN RANGE
acos	0 to PI
asin	-PI/2 to PI/2
atan	-PI/2 to PI/2
atan2	-PI/2 to PI/2

Since the tangent becomes very large for angles close to PI/2, the *atan2* function is often used to avoid computations with large numbers that might easily overflow. With *atan2*, you can express the large tangent value as a quotient of two more reasonable numbers.

The *cosh, sinh* and *tanh* routines compute the normal hyperbolic functions.

SEE

matherr

NAME

tzset Set time zone variable

SYNOPSIS

```
#include <time.h>

tzset();

/*   These symbols are defined in time.h:
*
* extern int daylight;
* extern long timezone;
* extern char *tzname[2];
* extern char tzstn[4];
* extern char tzdtn[4];
*
*/
```

DESCRIPTION

The *tzset* function assigns values to the time zone variables *daylight*, *timezone* and *tzname*. These variables are then used by *localtime* and other functions to correct from Greenwich Mean Time (GMT) to local time.

The values for these variables are obtained from the character string pointer named **_TZ**, which has the form:

```
char *_TZ = "aaabbbccc"
```

where **aaa** is the 3-letter abbreviation for the local standard time zone (e.g. CST), and **bbb** is a number from -23 to +24 indicating the value that is subtracted from GMT in order to obtain local standard time. Both **aaa** and **bbb** are required, but **ccc** is the abbreviation for the local daylight savings time zone (e.g. CDT), and it should be present only if daylight savings time is currently in effect. Initially, **_TZ** is set to NULL. It should be initialized with the address of a string corresponding to the correct time zone. If **_TZ** is NULL, *tzset* uses the default string **CDT6**.

When *tzset* is called, first *timezone* is loaded with the number of seconds that must be subtracted from GMT in orderd to get the local time. Next *daylight* is loaded with 0 if the **ccc** portion of **_TZ** is absent and 1 if **ccc** is present. Then the **aaa** and **ccc** parts are copied to **tzstn** and **tzdtn**, respectively, with null terminators. Finally, **tzname[0]** and **tzname[1]** are loaded with pointers to **tzstn** and **tzdtn**, respectively.

RETURNS

None.

> NOTE: The **tzstn** and **tzdtn** variables are Lattice extensions and will not exist on the typical UNIX system.

SEE

timedata, localtime

NAME

ungetc Push input character back

SYNOPSIS

```
#include <stdio.h>

r = ungetc(c, fp);
int r;                  return character or code
char c;                 character to be pushed back
FILE *fp;               file pointer
```

DESCRIPTION

This function pushes a character back to the specified Level 2 input file. The character need not be the same as the one that was most recently read. However, before calling *ungetc*, you must have read at least one character via *fgetc* or one of the other Level 2 input functions. Also, you can only push back one character; if you call *ungetc* more than once between input functions, the results are undefined.

RETURNS

Normally *ungetc* returns the character that was pushed back. However, if the end-of-file has been hit or if no characters have been read yet, the value EOF is returned.

EXAMPLE

```
#include <stdio.h>

main()
{
int c;

while(1)
  {
  printf("Loop 1...\n");
  while((c = getchar()) != EOF)
```

```
      {
      if(isalpha(c)) putchar(c);
      else break;
      }
    ungetc(c);
    printf("\n\nLoop 2...\n");
    while((c = getchar()) != EOF)
      {
      if(isalpha(c) == 0) putchar(c);
      else break;
      }
    ungetc(c);
    }
  printf("\n\nDone\n");
  }
```

NAME

utpack	Pack UNIX time
utunpk	Unpack UNIX time

SYNOPSIS

```
#include <stdlib.h>

ut = utpack(x);
utunpk(ut,x);

long ut;      packed UNIX time
char *x;      unpacked UNIX time
```

DESCRIPTION

These functions pack and unpack the 32-bit value time that is traditionally used in UNIX systems. This value is the number of seconds since 00:00:00, January 1, 1970. The *time* function returns the system clock in this form relative to Greenwich Mean Time.

The unpacked time is a 6-byte array in the following format:

BYTE		CONTENTS
x[0]	=>	year - 1970 (-128 to +127)
x[1]	=>	month (1 to 12)
x[2]	=>	day (1 to 31)
x[3]	=>	hour (0 to 23)
x[4]	=>	minute (0 to 59)
x[5]	=>	second (0 to 59)

Although this array is similar to the one produced by *getclk* and used by *stpdate*, note that the year is biased relative to 1970 instead of 1980. So, if you use *utunpk* followed by *stpdate*, you must subtract 10 from **x[0]** before the *stpdate* call. Note also that the year is a signed character and can be negative. A value of -3, for example, is 1967 (i.e. 1970 - 3).

SEE

ctime, getclk, gmtime, localtime, stpdate, time

EXAMPLE

```
/*
*
* Get a file time and convert it to UNIX time.
* No error checks.
*
*/
#include <time.h>
#include <dos.h>
#include <stdlib.h>

main(argc,argv)
int argc;
char *argv[ ];
{
char tt[6];
int fh;
long ft,ut;

ft = getft(argv[1]);
tt[0] -= 10;
ut = utpack(tt);
printf("File time is: %s\n",ctime(&ut));
}
```

NAME

_bufsize Level 2 I/O buffer size

SYNOPSIS

```
extern int _bufsiz;
```

DESCRIPTION

This external integer is used by the level 2 I/O system to determine the size of the buffers allocated for level 2 files. This location is also used to determine the size of a buffer attached to a file with the *setbuf* function. In this case, *_bufsiz* must be set to the size of the buffer before *setbuf* is called.

Note that the buffer is not allocated when the file is opened. Instead, the first I/O operation causes the buffer to be allocated from the local memory pool if one has not been previously specified with *setbuf.* This means that if *_bufsiz* is changed between the open call and the first I/O operation, the size of the buffer allocated for the file will be the value of *_bufsiz* at the time of the I/O operation, not the value when the file was opened.

SEE

fopen, setbuf

NAME

_fmode Default level 2 I/O mode

SYNOPSIS

```
extern int _fmode;
```

DESCRIPTION

This external integer is used by the *fopen* function to determine the translation mode to use when the programmer does not specify a mode in the *fopen* call. For AmigaDOS it is set to 0x8000, which specifies untranslated mode.

SEE

fopen

NAME

_FPERR Floating Point Error Code

SYNOPSIS

```
extern int _FPERR;
```

DESCRIPTION

This location will contain a non-zero value after any low-level floating point operation encounters an error. Low-level operations include addition, subtraction, multiplication, division, comparison, and conversion from one number representation to another (e.g., float to double).

The error codes and their corresponding symbols from **math.h** and **math.mac** are:

SYMBOL	VALUE	MEANING
FPEUND	1	Underflow
FPEOVF	2	Overflow
FPEDVZ	3	Divide by zero
FPENAN	4	Not a valid number
FPECOM	5	Not comparable

When the error occurs, the low-level operation passes the appropriate error code to *CXFERR*, which must store the code in *_FPERR*. Note that *_FPERR* is never reset by any low-level operation.

SEE

CXFERR, _FPA

EXAMPLE

```
/*
*
* This example performs uses the division operation
```

```
* to stimulate floating point errors.
*
**/
#include <math.h>
#include <stdio.h>
extern int _FPERR;
main()
{
double a,b,c;

while(feof(stdin) == 0)
  {
  printf("Enter divisor:  ");
  if(scanf("%lf",&a) != 1) exit(0);
  printf("Enter dividend: ");
  if(scanf("%lf",&b) != 1) exit(0);
  _FPERR = 0;
  c = b / a;
  printf("_FPERR = %d\n",_FPERR);
  printf("%e / %e = %e\n\n",b,a,c);
  }
}
```

NAME

_MNEED Minimum Dynamic Memory Needed

SYNOPSIS

```
extern long _MNEED;  to reference default value
long _MNEED = n;     to set initial value
```

DESCRIPTION

_MNEED indicates the minimum number of bytes that must be provided as the so-called *heap space* for use by the various memory allocators. This space, whose size is rounded up to be an even multiple of *_mstep*, is the initial block of memory in the local memory pool.

If one of the memory allocators later determines that it needs more heap space, it will call upon *sbrk* to request additional memory from AmigaDOS. This memory then becomes a non-contiguous addition to the local pool.

Each time *rbrk* is called, all the memory in the local pool is returned to the system, and a request for the amount of memory in *_MNEED* is made of AmigaDOS.

Because of this technique, you can use *_MNEED* to guarantee at the outset that your program will have enough heap space. However, if you don't do anything with *_MNEED*, the memory allocators will still try to honor your space requests until AmigaDOS indicates that no more memory is available.

There are two ways to initialize this item. If your space needs are known at compilation time, you can simply declare *_MNEED* as an initialized public symbol. That declaration will then override the default provided by the library, and the space will be obtained automatically by the startup routine (c.o). However, if you cannot determine your space needs until run time, you can load the appropriate value into *_MNEED* and then call *rbrk*. This, of course, should be only be done after all dynamic memory has been released via the appropriate de-allocator function. See the *rbrk* description for further information and cautions.

NAME

_main	Standard preprocessing for the main module
_tinymain	Special version of _main

SYNOPSIS

```
#include <stdlib.h>

_main(line);
_tinymain(line);

char *line;        /* ptr to command line that caused
execution */
```

DESCRIPTION

The *_main* function performs the standard preprocessing for the main module of a C program. It accepts a command line of the form:

```
pgmname arg1 arg2
```

and builds a list of pointers to each argument. The first pointer is to the program name. _main will also open the standard I/O files. (stdin, stdout, and stderr) _main calls the function *main* with the standard argc and argv parameters.

_tinymain is a special version of _main that does not open the standard I/O files. It can be used by specifying:

```
DEFINE _main=_tinymain
```

on the *BLINK* command line.

SEE

main

NAME

_MSTEP Memory Pool Increment Size

SYNOPSIS

```
extern int _MSTEP;
```

DESCRIPTION

This external integer is used by the memory allocation functions. It specifies the minimum amount of memory that will be allocated from the system for the local memory pool.

While additional memory is added to the local pool, it will not be contiguous with the memory already in the pool. If the additional amount is small, it can lead to severe fragmentation of the local pool. The memory allocation functions attempt to avoid this by rounding the amount needed up to the next multiple of the figure in *_MSTEP*.

This technique works well for small allocation requests. However, if your application requires mostly large blocks of memory, *_MSTEP* should be set to a small non-zero figure to allow for a more efficient allocation.

SEE

getmem, rlsmem

NAME

_OSERR DOS Error Information

SYNOPSIS

```
os_nerr       Number of AmigaDOS error codes
os_errlist    AmigaDOS error messages
#include <dos.h>
extern int _OSERR;
extern int os_nerr;
extern struct DOS_ERRS os_errlist[ ];
```

DESCRIPTION

The external integer named *_OSERR* contains error information returned by AmigaDOS after a system call has failed. In general, the Lattice library resets *_OSERR* at the beginning of any function that makes AmigaDOS system calls. Then if a system call fails during that function, the system error code is saved in *_OSERR*.

The AmigaDOS error number is mapped into an equivalent UNIX error number, which is placed in *errno*. If there is no appropriate UNIX number, *errno* will contain -1, defined symbolically as **EOSERR**. The function returns with a suitable error indication, which is usually -1 for functions that return integer values or NULL for functions that return pointers.

The **os_nerr** and **os_errlist** items are defined in a C source file named **oserr.c** and are used by the *poserr* function to print messages that correspond to the code found in *_OSERR*.

The following list applies to AmigaDOS 1.1 and is what we provide in **oserr.c**:

CODE	MEANING
103	Insufficient free store
104	Task table full
120	Argument line invalid or too long

121	File is not an object module
122	Invalid resident library during load
202	Object in use
203	Object already exists
204	Directory not found
205	Object not found
206	Invalid window
209	Packet request type unknown
210	Invalid stream component name
211	Invalid object lock
212	Object not of required type
213	Disk not validated
214	Disk write protected
215	Rename across devices attempted
216	Directory not empty
218	Device not mounted
219	Seek error
220	Comment too big
221	Disk full
222	File is protected from deletion
223	File is protected from writing
224	File is protected from reading
225	Not a DOS disk
226	No disk in drive
232	No more entries in directory

SEE

AmigaDOS Technical Reference Manual, poserr

NAME

_SLASH Directory separator character

SYNOPSIS

```
extern char _SLASH;
```

DESCRIPTION

This external character is used by various functions which construct file names. It specifies the character to be used for separating components of the directory path. For AmigaDOS, the default is a forward slash (/), while for MSDOS it is a backslash (\).

SEE

strmfn, strmfp

NAME

daylight	Daylight savings time flag
timezone	Timezone bias from GMT
tzname	Timezone names
tzstn	Standard time name
tzdtn	Daylight time name

SYNOPSIS

```
extern int daylight;
extern long timezone;
extern char *tzname[2];
extern char tzstn[4];
extern char tzdtn[4];
```

DESCRIPTION

These variables are initialized by the *tzset* function and are used by the *localtime* function to adjust from Greenwich Mean Time (GMT) to the local time.

The *daylight* item is non-zero if daylight saving time is currently in effect. The *timezone* value is the number of seconds that must be subtracted from GMT. The two *tzname* pointers point to *tzstn* and *tzdtn*, respectively. These strings contain the three-character names for standard time (*tzstn*) and daylight time (*tzdtn*).

SEE

localtime, tzset

Index

A

B

C

I

J

L

M

N

O

T

U

W

X

Y

NAME

closedir Terminate directory operation

SYNOPSIS

```
#include <sys/types.h>
#include <sys/dir.h>

void closedir(dfd)
DIR *dfd;    directory file descriptor
```

DESCRIPTION

This routine closes a directory previously opened with *opendir*. All memory and systems locks associated with the directory file descriptor are freed. You must call *closedir* for each descriptor opened with *opendir* in order to reclaim the memory and the lock associated with the descriptor.

SEE

opendir, readdir, telldir, seekdir

NAME

rewinddir	Restart directory operation
seekdir	Reposition directory operation

SYNOPSIS

```
#include <sys/types.h>
#include <sys/dir.h>

void seekdir(dfd, loc)
DIR *dfd;   dir file descriptor
long loc;   entry location

#define rewinddir(dfd) seekdir(dfd, (long)0)
```

DESCRIPTION

This routine changes the position from which *readdir* will read the next directory entry. dfd is a directory file descriptor returned from a successful call to *opendir*. loc is a directory entry location as returned from a call to *telldir*. NOTE: Seek locations are guaranteed only for the life of the dfd. If a directory is closed and reopened the directory entry locations may be invalid.

rewinddir resets the current position to the start of the directory.

SEE

opendir, readdir, closedir

NAME

isatty Test file descriptor for terminal

SYNOPSIS

```
#include <fcntl.h>

rc = isatty(fd)
int fd;              level 1 file descriptor
int rc;              return code
```

DESCRIPTION

This function takes a file descriptor as returned from a call to open and tests to see if the file descriptor is associated with a terminal device.

RETURNS

If the file descriptor is associated with a terminal device, the routine will return 1. Otherwise it will return 0.

SEE

open

NAME

stat Get file status

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

rc = stat(file, st)
char *file;          file name
struct stat *st;     stat info structure
int rc;              return code
```

DESCRIPTION

This function obtains information for the given file. If the file is not in the current directory the file path must be included as part of the file name. Permission to read, write, or execute the file is not required.

The information is placed into the stat structure pointed to by **st**. The structure is defined in **sys/stat.h** and contains information as follows:

```
struct   stat {
  dev_t           st_dev;      Amiga disk typ
  ino_t           st_ino;      Amiga disk key
  unsigned short  st_mode;     (file/directory)|permissions
  short           st_nlink;    number of links ;not used, 0
  uid_t           st_uid;      user id         ;not used, 0
  gid_t           st_gid;      group id        ;not used, 0
  dev_t           st_rdev;                     ;not used, 0
  off_t           st_size;     file size in bytes
  /* last access time; not supported by AmigaDos;        */
  /* same as st_mtime                                    */
  time_t          st_atime;
  /* create time; not supported by AmigaDos;             */
  /* same as st_mtime                                    */
  time_t          st_ctime;
  /* last modified time in seconds since Jan 1, 1978     */
  time_t          st_mtime;
  long            st_blksize;    size of one block: =512
```

```
long            st_blocks;     size of file in blocks
};
```

The following is a list of defines that are or'ed together to form the **st_mode** field.

Symbol	Meaning
S_IFMT	type of file
S_IFDIR	directory
S_IFREG	regular file
S_ISCRIPT	is a script
S_IPURE	executable
S_IARCHIVE	archive file
S_IREAD	read permission
S_IWRITE	write permission
S_IEXECUTE	execute permission
S_IDELETE	delete permission

RETURNS

If the operation is successful, the function returns 0. Otherwise it returns -1 and places error information in **errno** and **_OSERR**.

SEE

errno, _OSERR, chmod

Class: UNIX

NAME

opendir   Initiate directory operation

SYNOPSIS

```
#include <sys/types.h>
#include <sys/dir.h>

dfd = opendir(dirname)
char *dirname;        directory name
DIR  *dfd;            return directory file descriptor
```

DESCRIPTION

Given a directory name, this routine opens a directory for subsequent read access.

RETURNS

If the open succeeds it will return a pointer to a handle that contains the following information.

```
typedef struct _dirdesc {
        long   dd_fd;      /* system directory lock   */
        long   dd_loc;     /* current directory posn  */
        long   dd_size;    /* size of dd_buf in bytes */
        char  *dd_buf;     /* system structure info   */
        } DIR;
```

Otherwise **NULL** is returned.

SEE

readdir, seekdir, telldir, closedir

EXAMPLE

```
/* The following is an example of how to open and search */
/* the contents of a directory for a particular entry.   */
#include <sys/types.h>
#include <sys/dir.h>
#include <string.h>

searchdir(dname, file)
char *dname,   /* directory name */
     *file;    /* file name      */
{
int rc;
DIR *dfd;      /* directory descriptor */
struct direct *dptr; /* dir entry      */

rc = 0;
dfd = opendir(dname);

while ((dptr = readdir(dfd)) != NULL)
   {
   if (!strcmp(file, dptr->d_name))
      {
      rc = 1; /* Found a match */
      break;
      }
   }
closedir(dfd);
return(rc);
}
```

NAME

telldir   Get directory position

SYNOPSIS

```
#include <sys/types.h>
#include <sys/dir.h>

loc = telldir(dfd)
DIR *dfd;      directory file descriptor
long loc;      current read position
```

DESCRIPTION

This routine returns the current read position for the given dfd. This position is where *readdir* would obtain the next directory entry if it was called. **loc** is also the value that you would pass to *seekdir* if you wanted to return directly to this same position.

> loc values are good only for the life of the dfd. If you close a directory and reopen it, the loc values are invalidated.

RETURNS

The location of the next directory entry is returned.

SEE

opendir, readdir, seekdir, closedir

NAME

readdir Read directory element

SYNOPSIS

```
#include <sys/types.h>
#include <sys/dir.h>

dptr = readdir(dfd)
DIR *dfd;              directory file descriptor
struct direct *dptr;  pointer to a directory entry
```

DESCRIPTION

This function returns the next directory entry for a given directory. **dfd** is the directory file desciptor obtained from a call to *opendir*, and **dptr** is the directory entry. A directory entry has the following format.

```
struct direct {
  u_long  d_ino;     /* Amiga disk key                  */
  u_short d_reclen;  /* length of this record           */
  u_short d_namlen;  /* dynamic length of d_name        */
  off_t   d_off;     /* offset of disk directory entry  */
  char    d_name[MAXNAMLEN + 1]; /* maximum name length */
};
```

RETURNS

A pointer to a **struct direct** which is the next directory entry. If the end of the directory list is reached or if an error occurs a NULL is returned. For errors, the error information is placed in **errno** and **_OSERR**.

SEE

opendir, seekdir, telldir, closedir

Master Index

Master Index

Master Index

Lattice C Master Index

Master Index for the Lattice Amiga C Compiler Manual

Lattice, Inc.
2500 S. Highland Avenue
Lombard, IL 60148
USA

A Subsidiary of SAS Institute Inc.

Lattice C Index

This manual was formatted using **HighStyle®** by Lattice, Inc.

SAS/C® Master Index

Master Index for the SAS/C ® Compiler Manual

SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513-2414
USA

SAS/C®Master Index

This document was produced using ***HighStyle***®, the Lattice Document Composition System.

Master Index

A

B

C

E

P

Q

R

S

T

Lattice

Lattice C Compiler For Amiga

Debugger
Library Reference
Master Index

Volume 2